## VAX PASCAL User's Guide

Order Number: AA-H485D-TE

#### March 1985

This document describes how to interact with the VAX/VMS operating system using VAX PASCAL. It contains information dealing with input and output with RMS, optimizations, program section use, calling conventions, and error processing. This document is intended for programmers who have full working knowledge of PASCAL.

**Revision/Update Information:** 

This revised document will supersede information in the VAX-11 PASCAL

User's Guide AA-H485C-TE

**Operating System and Version:** 

VAX/VMS Version 4.0 or later

Software Version:

VAX PASCAL V3.0

digital equipment corporation maynard, massachusetts

#### First Printing, November 1979 Revised, March 1985

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright @1979, 1985 by Digital Equipment Corporation

All Rights Reserved. Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	digit

ZK-2616

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

# **Contents**

PREFA	ACE		xii
CHAPTER 1	VAX P	ASCAL SYSTEM ENVIRONMENT	1-1
1.1	USE OF	PROGRAM SECTIONS	1-1
	1.1.1	Default Program Sections	_ 1-2
	1.1.2		
	1.1.3		
	1.1.4	Establishing Program Section Properties	_ 1-4
1.2	STORA	GE ALLOCATION	1-6
	1.2.1	Allocation of Variables	1-6
	1.2.2	Allocation of Symbolic Constants and Executable Blocks	_ 1-7
	1.2.3		
1.3	ALLOCA	ATION SIZES OF VARIABLES	1-10
1.4	ALIGNN	MENT BOUNDARIES	1-16
1.5	REPRES	SENTATION OF VARYING DATA	1-18
1.6	REPRES	SENTATION OF FLOATING-POINT DATA	1-19
	1.6.1	Single-Precision (SINGLE, REAL Types)	_ 1-19
	1.6.2	Double-Precision (DOUBLE Type)	_ 1-21
		1.6.2.1 D_Floating Point • 1-21	
		1.6.2.2 G_Floating-Point • 1-22	
	1.6.3	Quadruple-Precision (QUADRUPLE Type)	_ 1-23

CHAPTER 2	PROGR	AM OPTIMIZATION AND EFFICIENCY	2-1
2.1	COMPI	ER OPTIMIZATIONS	2-1
	2.1.1	Compile-Time Evaluation of Constants	2-3
	2.1.2	Elimination of Common Subexpressions	2-4
	2.1.3	Elimination of Unreachable Code	
	2.1.4	Code Hoisting from Structured Statements	
	2.1.5	Inline Code Expansion for Predeclared Functions	
	2.1.6	Inline Code Expansion for User Declared Routines	
	2.1.7	Rearranging Operations	
	2.1.8	Partial Evaluation of Logical Expressions	
	2.1.9	Value Propagation	
	2.1.10	Alignment of Compiler-Generated Labels	2-10
	2.1.11	Reducing Errors Through Optimization	
2.2	PROGR	AMMING CONSIDERATIONS	2-11
2.3	ОРТІМІ	ZATION CONSIDERATIONS	2-13
	2.3.1	Subexpression Evaluation	2-13
	2.3.2	Lowest Negative Integer	
	2.3.3	Pointer References	2-14
	2.3.4	Variant Records	2-15
	2.3.5	Type Cast Operations	
	2.3.6	Effects of Optimization on Debugging	
CHAPTER 3	CALLIN	IG CONVENTIONS	3-1
3.1	VAY DD	OCEDURE CALLING STANDARD	3-1
3.1	3.1.1	Parameter Lists	
	3.1.1		
	3.1.3	Contents of the Call Stack	
	5.1.5	Contents of the Call Stack	ა-ა
3.2	PASSIN	G PARAMETERS TO EXTERNAL ROUTINES	3-7

	By-Reter	ence Mechanism
		Value Semantics • 3-10
	3.2.1.2	Variable Semantics • 3-11
		%REF Mechanism Specifier • 3-12
	3.2.1.4	[REFERENCE] Attribute • 3-13
3.2.2	By-Imme	ediate-Value Mechanism
	3.2.2.1	%IMMED Mechanism Specifier • 3-13
	3.2.2.2	[IMMEDIATE] Attribute • 3-14
3.2.3	By-Desc	riptor Mechanism
	3.2.3.1	
	3.2.3.2	%DESCR Mechanism Specifier • 3-19
	3.2.3.3	
	3.2.3.4	CLASS_A and CLASS_NCA Attributes • 3-20
3.2.4	Mechani	sm Specifiers on Actual Parameters
1 Adding		
CALLIN	G VAX/VM	S SYSTEM SERVICES
CALLIN 3.4.1	G VAX/VMS Passing	S SYSTEM SERVICES Parameters to System Services
CALLIN 3.4.1 3.4.2	G VAX/VM3 Passing Data Str	S SYSTEM SERVICES Parameters to System Services ucture Parameters
CALLIN 3.4.1 3.4.2 3.4.3	G VAX/VM3 Passing Data Str Default	S SYSTEM SERVICES Parameters to System Services ucture Parameters Parameters
	G VAX/VMS Passing Data Str Default I Calling S	S SYSTEM SERVICES Parameters to System Services ucture Parameters Parameters System Services as Procedures
CALLIN 3.4.1 3.4.2 3.4.3 3.4.4	G VAX/VMS Passing Data Str Default I Calling S	S SYSTEM SERVICES Parameters to System Services ucture Parameters Parameters
CALLIN 3.4.1 3.4.2 3.4.3 3.4.4 3.4.5	G VAX/VMS Passing Data Str Default I Calling S System	S SYSTEM SERVICES Parameters to System Services ucture Parameters Parameters System Services as Procedures
3.4.1 3.4.2 3.4.3 3.4.4 3.4.5	G VAX/VMS Passing Data Str Default I Calling S System TING THE S	S SYSTEM SERVICES Parameters to System Services ucture Parameters Parameters System Services as Procedures Service Example

CHAPTER 4	INPUT	AND OUT	PUT WITH RMS	
4.1	RMS FII	LE CHARA	CTERISTICS	
	4.1.1		anization	
		4.1.1.1		
		4.1.1.2	Relative Organization • 4-3	
			Indexed Organization • 4-3	
	4.1.2	Record A	Access	
		4.1.2.1		
		4.1.2.2		
		4.1.2.3	Keyed Access ● 4-5	
4.2	RMS RE	CORD FOR	RMATS	
	4.2.1	Fixed-Le	ngth RMS Records	
	4.2.2		Length RMS Records	
	4.2.3		Record Format	
4.3	INPUT/0	OUTPUT EF	RROR DETECTION	
4.4	OPEN P	ROCEDURI	E PARAMETERS	
	4.4.1	File Vari	able	
	4.4.2		ne	
	4.4.3			
	4.4.4		ength	
	4.4.5		Method	
	4.4.6		ype	
	4.4.7		Control	
	4.4.8		tion	
	4.4.9		on	
	4.4.10			
	4.4.11		ion	
	4.4.12		File Name	
	4.4.13		covery	
4.5	CLOSE F	PROCEDUR	E PARAMETERS	
-	4.5.1		on	
	4.5.2	•	ion	
	4.5.3		covery	

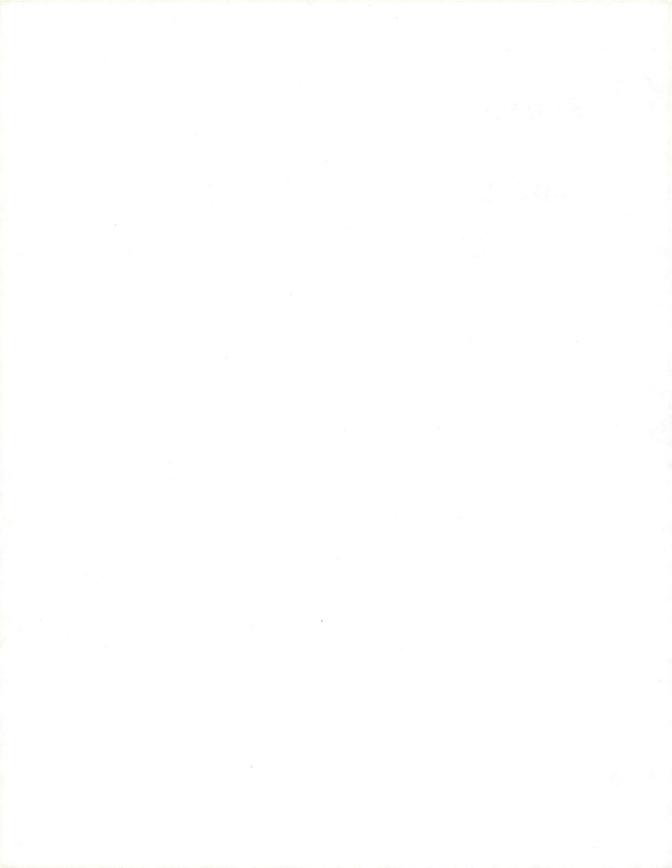
	4.6	FILE SH	ARING	4-2
	4.7	RECOR	DLOCKING	4-2
	4.8	HEING	INDEXED FILES	4-2
	4.0	4.8.1		4-2
		4.8.1	Current Component and Next Component Pointers	
		4.8.2	Writing Indexed Files	
		4.8.4	Reading Indexed Files	
		4.8.5	Updating Components	
		4.8.6	Deleting Components	
		4.8.7	Exception Conditions	4-3
	4.9	INPUT/0	OUTPUT CONSIDERATIONS ON TEXT FILES	4-:
		4.9.1		4-3
		4.9.2	The state of the s	4-3
		4.9.3		
		4.9.4	Delayed Device Access to Text Files	
	4.10	INTERP	ROCESS COMMUNICATION: MAILBOXES	4-
	4.11	СОММ	JNICATING WITH REMOTE COMPUTERS: NETWORKS	4-
СНА	PTER 5	ERROR	PROCESSING AND CONDITION HANDLERS	5
	5.1	RUN-TII	ME LIBRARY DEFAULT ERROR PROCESSING	5
	5.2	DEFINIT	TIONS OF TERMS	5
	5.3	OVERVI	EW OF VAX CONDITION HANDLING	5
		5.3.1	Condition Signals	5
		5.3.2	Handler Responses	5
	5.4	WRITIN	IG CONDITION HANDLERS	5
		5.4.1	Establishing and Removing Handlers	5

	5.4.2	Parameters for Condition Handlers	5-7
	5.4.3	Handler Function Return Values	5-9
	5.4.4	Condition Values and Symbols	5-10
5.5	HANDLI	NG FAULTS AND TRAPS	5-11
5.6	EXAMPI	LES OF CONDITION HANDLERS	5-12
APPENDIX A	DIAGN	IOSTIC MESSAGES	A-1
A.1	COMPIL	ER DIAGNOSTICS	A-1
A.2	RUN-TIN	ME DIAGNOSTICS	A-56
APPENDIX B FUNCTION		RS RETURNED BY THE STATUS AND STATUSV	B-1
APPENDIX C PASCAL	DECLA	ARING RUN-TIME LIBRARY PROCEDURES IN VAX	C-1
APPENDIX D	ENTRY	POINTS TO PASCAL UTILITIES	D-1
D.1	PAS\$FA	B (F)	D-1
D.2	PAS\$RA	B(F)	D-2
D.3	PAS\$MA	ARK2 (S)	D-3
D.4	PAS\$RE	LEASE2 (P)	D-4

APPENDIX E VERSION		ENCES BETWEEN VERSION 1 AND SUBSEQUENT	E-1
E.1	DECOM	MITTED FEATURES	E-2
	E.1.1	Dynamic Array Parameters	E-2
	E.1.2		E-3
	E.1.3	Printing Hexadecimal and Octal Values	E-4
	E.1.4		E-5
	E.1.5	Specifying Qualifiers in the Source Code	E-6
E.2	/OLD_VE	ERSION QUALIFIER	E-7
	E.2.1	Comment Delimiters	E-8
	E.2.2	%INCLUDE Files	E-8
	E.2.3	Multidimensional Packed Arrays	E-8
	E.2.4	Storage of Components	E-9
	E.2.5	Storage of Sets	E-9
	E.2.6		E-10
	E.2.7	MOD Operator	E-10
	E.2.8	String Variable Parameters to the READ Procedure	E-10
	E.2.9	Field Widths	E-11
	E.2.10	Global Identifiers	E-11
	E.2.11	Allocation in Program Sections	E-11
E.3	MINOR	LANGUAGE CHANGES	E-12
APPENDIX F	EXAMI	PLES OF USING SYSTEM SERVICES	F-1
F.1	CALLING	RMS PROCEDURES	F-1
F.2	SYNCHE	RONIZING PROCESSES USING AN AST ROUTINE	F-3
F.3	ACCESS	SING DEVICES USING SYNCHRONOUS INPUT/OUTPUT	F-6
F.4	сомми	UNICATING WITH OTHER PROCESSES	F-7

	F.5	SHARING CODE AND DATA	F-12
	F.6	GATHERING AND DISPLAYING DATA AT TERMINALS	F-14
	F.7	CREATING, ACCESSING, AND ORDERING FILES	F-19
	F.8	MEASURING AND IMPROVING PERFORMANCE	F-26
	F.9	ACCESSING HELP LIBRARIES	F-27
	F.10	CREATING AND MANAGING OTHER PROCESSES	F-30
	F.11	TRANSLATING LOGICAL NAMES	F-34
IND	EX		
FIGU	JRES		<del></del>
	1–1:	Storage of VARYING Data	1-19
	1–2:	Single-Precision Floating-Point Data Representation	1-20
	1–3:	D_Floating-Point Double-Precision Representation	1-21
	1-4:	G_Floating-Point Double-Precision Representation	1-22
	1–5:	Quadruple-Precision Floating-Point Representation	1-23
	3–1:	Contents of the Run-Time Stack	3-5
TAB	LES		
	1–1:	Program Section Properties	1-2
	1-2:	Program Section Data	
	1–3:	Required Program Section Properties	
	1-4:	Storage of Types	
	3–1:	Function Return Methods	
	3-2:	Foreign Mechanism Parameters	
	3-3:	Parameter Descriptors	
	3–4:	Passing Parameters to System Services	

4–1:	Valid Combinations of Record Access Method and File Organization	4
4–2:	Summary of OPEN Procedure Parameters	
4–3:	Carriage-Control Characters	
B-1:	STATUS and STATUSV Return Values	
C-1	Access Type Translations	(
C-2	Data Type Translations	(
C-3	Passing Mechanism Translations	
C-4	Parameter Form Translations	(
E-1	Summary of Version 1 OPEN Parameters	



# **Preface**

## **Intended Audience**

This manual is designed for programmers who have full working knowledge of PASCAL. It describes how to interact with the VAX/VMS operating system using VAX PASCAL.

This manual does not describe detailed language information, or information on how to create and run programs.

## **Structure of This Document**

This manual consists of the following chapters and appendixes:

- Chapter 1 provides information on program section use, storage allocation, and data representations
- Chapter 2 explains optimizations performed by the VAX PASCAL compiler and techniques for writing efficient programs and modules
- Chapter 3 describes conventions followed in calling procedures
- Chapter 4 provides information on input and output operations
- Chapter 5 discusses error processing, in particular, how to use the VAX/VMS condition-handling facility
- Appendix A lists complete run- and compile-time error messages
- Appendix B lists errors detected by the STATUS function
- Appendix C provides information on how to declare run-time library procedures in VAX PASCAL
- Appendix D describes the entry points to utilities in the VAX Run-Time Library that can be called as external VAX PASCAL routines
- Appendix E discusses the differences between VAX PASCAL Version 1 and all subsequent versions of VAX PASCAL
- Appendix F supplies examples of using system services

## **Associated Documents**

- Programming in VAX PASCAL provides detailed language information and information on how to create and run VAX PASCAL programs.
- Installing VAX PASCAL provides information on how to install VAX PASCAL on your VMS operating system.
- The manuals that accompany the operating system provide full information about VAX/VMS. The VAX/VMS Master Index briefly describes all VAX/VMS system documentation, defines the intended audience for each manual, and provides a synopsis of each manual's contents.

# **Conventions Used in This Document**

This document uses the following conventions.

Convention	Meaning
{ }	Large braces enclose lists from which you must choose one item; for example:
,	<pre>{ expression }     statement }</pre>
	A horizontal ellipsis means that the item preceding the ellipsis can be repeated; for example:
	digit
{}	Braces followed by a comma and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with commas; for example:
	{label},

Convention	Meaning
( );	Braces followed by a semicolon and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with semicolons; for example:
	REPEAT {statement}; UNTIL expression
	A vertical ellipsis in a figure or example means that not all of the statements are shown.
[]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays, sets, and attribute lists; for example:
	ARRAY[index1]
[]	Double brackets enclose items that are optional; for example:
	EOLN [[file-variable]]
Items in UPPERCASE letters and special symbols	Uppercase letters and special symbols in syntax descriptions indicate VAX PASCAL reserved words and predeclared identifiers; for example:
	BEGIN END
Items in lowercase letters	Lowercase letters represent elements that you must replace according to the description in the text.

In addition, the following notation is used to denote special nonprinting characters:

Space character

In this manual, complex examples and syntax diagrams have been divided into several lines to make them easy to read. PASCAL does not require that you format your programs in any particular way; therefore, you should not regard the formats used in this manual as mandatory.

# **VAX PASCAL System Environment**

This chapter describes the relationship between the VAX/VMS operating system and the VAX PASCAL compiler. It covers the following topics:

- Program sections
- Storage allocation of variables, constants, and blocks
- Storage allocation sizes of variables of all VAX PASCAL data types
- Alignment boundaries for variables of all VAX PASCAL data types
- Data representation of VARYING OF CHAR and floating-point data

# 1.1 Use of Program Sections

The VAX PASCAL compiler uses contiguous areas of memory, called program sections, to store information about a program. The VAX Linker controls memory allocation and sharing according to the properties of each program section. (The VAX/VMS Linker Reference Manual refers to the various characteristics of program sections as "attributes." This chapter uses the term "properties" to avoid confusion with the VAX PASCAL attribute classes.) Table 1-1 lists the possible program section properties.

Table 1-1: Program Section Properties

Class Meaning	
PIC/NOPIC	Position independent or position dependent
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute
GBL/LCL	Global or local scope
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable
WRT/NOWRT	Writeable or nonwriteable
SHR/NOSHR	Shareable or nonshareable

When constructing an executable image, the linker divides the image into sections. Each image section contains program sections that have the same properties. The linker controls memory allocation by arranging image sections according to program section properties. You can use special linker options to change program section properties and to influence the memory allocation in the image. You include these options in an options file, which is input to the linker. The options and the options file are described in the *VAX/VMS Linker Reference Manual*.

## 1.1.1 Default Program Sections

The VAX PASCAL compiler establishes as many as three program sections, \$CODE, \$LOCAL, and PAS\$GLOBAL. Table 1–2 summarizes the kinds of data contained in these sections by default.

Table 1-2: **Program Section Data** 

Default Program Section	Data
\$CODE	Machine instructions; constants needing storage; nonvolatile, READONLY, static variables
\$LOCAL	Writeable variables declared with the STATIC attribute; writeable variables that use default static allocation and are declared at program or module level of a nonoverlaid compilation unit
PAS\$GLOBAL	Writeable variables that use default allocation and are declared at program or module level of an OVERLAID compilation unit

These three default program sections are established only when the program requires them. For example, unless the program contains an OVERLAID compilation unit, PAS\$GLOBAL is not created. If the program does not include any static variables, \$LOCAL is not declared. And if all necessary program sections are user-defined, the compiler does not establish any of the default program sections.

## 1.1.2 PSECT Attribute

In a VAX PASCAL program, you can control the allocation of virtual address space by establishing the program section in which storage for a variable, routine, or compilation unit should be allocated. This control is provided by the PSECT attribute. The PSECT attribute has the following form:

PSECT(identifier)

#### identifier

The name of the program section. This name can designate either a program section that is established by the compiler or one that is created by the user.

The PSECT attribute is useful for placing static variables and executable blocks in program sections that will be shared among executable images. For example, if several programs access the same error-processing routines, you could group the routines in a separate program section. Since these routines are likely to be called only to handle unusual conditions, the image section containing the executable code for them need not be paged into physical memory until one of the routines is called. Then the image section is paged into main memory until the routine has completed execution.

#### 1.1.3 COMMON Attribute

In a VAX PASCAL program, you can allocate storage for a variable in an overlaid program section called a common block. This capability is provided by the COMMON attribute. The COMMON attribute has the following form:

COMMON [ (identifier) ]

#### identifier

The name of a common block.

When no identifier is present, the name of the common block is the same as the variable having the COMMON attribute.

Only one variable can be allocated in a particular common block. By storing variables in common blocks, a VAX PASCAL program can share variables with programs written in other VAX languages, such as FORTRAN, PL/I, COBOL, and BASIC.

## 1.1.4 Establishing Program Section Properties

Whether the compiler establishes a program section, or you create one, the program section gets assigned one property from each class listed in Table 1–1. These properties are assigned to satisfy the requirements of the variables and executable blocks that have been allocated in the same program section. Table 1–3 lists the properties required by various objects.

Table 1-3: **Required Program Section Properties** 

Object	Properties				
	EXE	RD	WRT	NOSHR	
READONLY Variable		Χ			
WRITEONLY Variable		Χ	Χ	$X^1$	
Read/write Variable		X	Χ	X <sup>1</sup>	
Executable Block	X	X			

<sup>&</sup>lt;sup>1</sup>Unless the variable has a COMMON attribute

All program sections are position independent and relocatable; all except those established by the COMMON attribute are concatenated and local. Program sections established by COMMON are overlaid and global. The remaining properties are assigned as follows:

- 1. The first time the compiler encounters the name of a particular program section (including a common block), it initializes the program section to be readable, nonwriteable, shareable, and nonexecutable. Thus, the program section's initial properties are GBL, NOEXE, NOWRT, OVR, PIC, RD, REL, and SHR.
- 2. If storage for a writeable object is allocated in the same program section, the program section instantly becomes writeable and if it is not a common block, nonshareable. Thus, the program section's write property changes from NOWRT to WRT and, if the program section is not a common block, its share property changes from SHR to NOSHR.

Note that writeable and nonshareable are complementary properties; that is, when a program section is assigned the WRT property, it immediately loses the SHR property it had initially. A COMMON block, however, always retains the SHR property.

If you want to guarantee that READONLY variables can never be modified, you can allocate storage exclusively for them in a separate program section that will always remain nonwriteable.

## 1.2 Storage Allocation

The next two sections discuss storage allocation of variables, symbolic constants, and executable blocks. The last section gives examples.

### 1.2.1 Allocation of Variables

When allocating storage for a variable, the compiler first determines an allocation attribute for the variable. If the allocation attribute is STATIC or COMMON, the compiler then chooses a program section in which to allocate storage. The compiler applies the following rules sequentially to determine the allocation attribute:

- 1. If the variable is declared with an allocation attribute, the attribute specifies the variable's allocation.
- 2. If the variable is declared with a visibility attribute other than LOCAL, its allocation is STATIC.
- 3. If the variable is declared in a routine, its allocation is AUTOMATIC.
- 4. If the variable is declared at the outermost level of a module, its allocation is STATIC.
- 5. If the variable is declared at the outermost level of a program, the compiler must choose between STATIC and AUTOMATIC allocation. Whenever possible, the compiler uses automatic allocation for variables that are referred to only in the body of the main program because automatic allocation is more efficient. The compiler uses static allocation if the variable is VOLATILE, initialized at its declaration, or referred to in a nested block, or if the program has an ENVIRONMENT or OVERLAID attribute. Because program-level variables may be statically allocated, VAX PASCAL does not support recursive calls on the main program block.

The compiler applies the following rules sequentially to choose the program section in which to allocate storage for nonexternal COMMON and STATIC variables:

- If the variable has a COMMON attribute, storage is allocated in a common block that has either the same name as the variable, or the name specified by the identifier that accompanies the attribute.
- If the variable has a PSECT attribute, the identifier that accompanies the attribute supplies the name of the program section in which storage is to be allocated.

- If the variable does not have a COMMON, PSECT, or STATIC attribute, but is declared at the outermost level of an OVERLAID compilation unit, storage is allocated in the program section PAS\$GLOBAL.
- If the variable has the READONLY attribute and has neither a PSECT nor a COMMON attribute, its storage is allocated in the program section in which storage for executable code is currently being allocated (see Section 1.2.2).
- All other static variables are allocated in the program section \$LOCAL.

#### 1.2.2 Allocation of Symbolic Constants and Executable Blocks

When allocating storage for symbolic constants and executable blocks, the compiler determines the appropriate program section by applying the same rules of scope to program section names that it applies to identifiers. The compiler always allocates storage in the program section whose name appeared in the most recent heading of a routine or compilation unit.

When a program begins, the compiler establishes the \$CODE program section. Storage is allocated in \$CODE for symbolic constants and executable blocks unless a PSECT attribute appears in the heading of a routine or compilation unit and directs that storage be allocated in a different program section.

If a routine has the INITIALIZE attribute, the address of its entry mask is stored in a program section named LIB\$INITIALIZE. The properties of this program section, which include NOPIC, are discussed in the VAX/VMS Run-Time Library Routines Reference Manual.

#### 1.2.3 **Examples**

The following examples illustrate how the compiler establishes program sections to allocate storage for symbolic constants, variables, and executable blocks. The comments in the program indicate the names of the program sections used.

#### **Examples**

```
    PROGRAM Order_Process (INPUT, OUTPUT);

                                                     (* $LOCAL *)
      Data_Record : RECORD
                    Account_Number : INTEGER;
                    Order_Number : INTEGER;
                    Name : VARYING[60] OF CHAR;
                    Address : VARYING [40] OF CHAR;
                    Phone : PACKED ARRAY[1..7] OF CHAR;
                    Total_Amount : REAL;
                    END;
   CONST
                                                    (* $CODE *)
      Message_String = 'No Invalid Data';
      [PSECT(Error_Sect)] PROCEDURE Account_Error;
                                                     (* $LOCAL *)
            Account_Array : [STATIC] ARRAY[1..15] OF INTEGER;
         CONST
            Error_String = 'Invalid Account Number';
         PROCEDURE Order_Error;
            VAR
                                              (* Automatic storage *)
               Order_Array : ARRAY[1..10] OF INTEGER
            CONST
                                                     (* Error_Sect *)
               Error_String = 'Invalid Order Number';
            BEGIN (* Procedure Order_Error *)
            Data_Record.Order_Number := 0;
                                                     (* Error_Sect *)
            END;
                    (* Procedure Order_Error *)
         PROCEDURE Amount_Error;
            BEGIN
                    (* Procedure Amount_Error *)
                                                    (* Error_Sect *)
            END:
                    (* Procedure Amount_Error *)
         BEGIN
                    (* Procedure Account_Error *)
                                                    (* Error_Sect *)
         END;
                   (* Procedure Account_Error *)
```

```
BEGIN (* Main Program *)
. (* $CODE *)
. END. (* Main Program *)
```

This example illustrates how the compiler allocates storage in user-created program sections in conjunction with the default VAX PASCAL program sections. Storage for all variables with static allocation is allocated in \$LOCAL. Storage for the executable block of the main program and the symbolic constant Message\_String is allocated in \$CODE. Storage is allocated in the user-created program section, Error\_Sect, for the symbolic constant Error\_String and the executable block of the procedure Account\_Error. Because the procedures Order\_Error and Amount\_Error fall within the scope of Account\_Error, storage is allocated in the same program section, Error\_Sect, for their symbolic constants and executable blocks. Storage for Data\_Record is in \$LOCAL because it was referred to in a nested block.

```
[OVERLAID] MODULE Over_Mod;
   Mod Var : INTEGER:
                                        (* PAS$GLOBAL *)
                                       (* PAS$GLOBAL *)
   Real Var : REAL:
   Static_Var : [STATIC] INTEGER;
                                      (* $LOCAL *)
PROCEDURE Block1
   (VAR Block_Param : REAL);
                                        (* $LOCAL *)
     Block_Var : [STATIC] UNSIGNED;
                                        (* $CODE *)
   CONST
      Block_Const = 'Nested Constant';
                                        (* $CODE *)
   BEGIN (* Procedure Block1 *)
   END: (* Procedure Block1 *)
END.
         (* MODULE Over_Mod *)
```

Because Over\_Mod is an OVERLAID compilation unit, storage is allocated in PAS\$GLOBAL for the module-level variables Mod\_Var and Real\_Var. However, the variables Block\_Var and Static\_Var are declared with the STATIC attribute; therefore, storage is allocated for them in \$LOCAL. Storage is allocated in \$CODE for Block\_Const and the executable block of the nested procedure Block1.

#### MODULE Normal\_Mod;

```
TYPE
  Read_Int = [READONLY.PSECT(Read_Var)] INTEGER:
  Data_File, Answer_File : TEXT;
                                                 (* $LOCAL *)
  Add_Value : Read_Int;
                                                 (* Read_Var *)
  Mult_Value, Factor : [READONLY, PSECT(Read_Var)] REAL;
                                                 (* Read_Var *)
  Div_Value : [READONLY, PSECT(Read_Var)]DOUBLE;
                                                 (* Read_Var *)
                                                 (* $LOCAL *)
  In_Value : INTEGER:
  Result : DOUBLE:
                                                 (* $LOCAL *)
END. (* Module Normal_Mod *)
```

The TYPE section in this example defines the type Read\_Int and gives it the READONLY and PSECT attributes. The identifier Read\_Var specifies that storage for variables of type Read\_Int is to be allocated in the user-created program section Read\_Var. The VAR section declares four READONLY variables and specifies that storage for them is to be allocated in Read\_Var. If no other variables are allocated in Read\_Var, the program section will retain the properties NOEXE, NOWRT, RD, and SHR throughout the program. Storage for the other four variables declared in the VAR section is allocated in \$LOCAL since they are at the outermost level of a MODULE.

## 1.3 Allocation Sizes of Variables

The VAX PASCAL compiler allocates storage for variables (and components of structured variables) when they are declared. For every VAX PASCAL data type, the compiler knows the allocation size required when a variable of the type occurs in either an unpacked or a packed context. The unpacked size is always represented in bytes, while the packed size is represented in bits.

The packed size of a variable is the minimum number of bits required to represent all values of the variable's type. In general, the compiler uses the "32-bit rule" to determine the allocation size for a component of a structured variable:

A component whose length is 32 bits or fewer is packed into as few bits as possible and may be UNALIGNED.

A component whose length is greater than 32 bits is allocated the smallest number of bytes possible and must be aligned on a byte boundary.

If one of the size attributes (BIT, BYTE, WORD, LONG, QUAD, or OCTA) is applied to the variable, the size specified by the attribute represents the variable's packed size. If no size attribute is applied to the variable, the compiler calculates the unpacked size so that it is structurally compatible with the base type of the variable's type.

Table 1-4 illustrates the allocation size for variables of each type when they occur in either a packed or an unpacked context.

Table 1-4: **Storage of Types** 

Data Type	Unpacked Size in Bytes	Packed Size in Bits	
INTEGER	4		
UNSIGNED	4	32	
CHAR	1	8	
BOOLEAN	1	1	
Enumerated 1, if 256 elements or fewer; 2, if more than 256 elements elements: 65,535)		log2(number of elements) 1 + 1	

<sup>&</sup>lt;sup>1</sup>This is known as the ceiling function, where the smallest integer is greater than or equal to X.

Table 1-4: (Cont.) Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits  Minimum number in which upper and lower bounds can be expressed <sup>2</sup>	
Subrange	Size of base type		
REAL/SINGLE	4	32	
DOUBLE	8	64	
QUADRUPLE	16	128	
Pointer 4		32	
Unpacked ARRAY	Sum of unpacked sizes in bytes of all components, plus the sum of the sizes in bytes of any holes created to meet alignment requirements	(Unpacked size in bytes) * 8	

 $<sup>^2</sup>$  The formula to compute the size of a packed subrange is MAX (X,Y) + Z where X, Y, and Z are computed as follows (LOW represents the low bound of the subrange; HIGH represents the upper bound):

```
IF HIGH > 0
IF LOW < -1
THEN
                                 THEN
                                 Y := [log2(HIGH)] + 1
ELSE
 X := [log2(-LOW - 1)] + 1
ELSE
                                   Y := 0;
  X := 0;
IF LOW >= 0
THEN
  Z := 0
ELSE
  Z := 1;
```

Table 1-4: (Cont.) Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits	
Unpacked RECORD	Sum of unpacked sizes in bytes of the fields in the fixed part and the largest variant, plus the sum of the sizes in bytes of any holes created to meet alignment requirements	(Unpacked size in bytes) * 8	
PACKED ARRAY	Sum of packed sizes in bits of all components, plus sum of sizes in bits of any holes created to meet alignment requirements; this sum is rounded up to a multiple of 8 and then divided by 8	Sum of packed sizes in bits of all components, plus sum of sizes in bits of any holes created to meet alignment requirements; if sum > 32, sum is rounded up to the next multiple of 8	
PACKED RECORD	Sum of packed sizes in bits of all fields in the fixed part and the largest variant, plus sum of sizes in bits of any holes created to meet alignment requirements; this sum is rounded up to a multiple of 8 and then divided by 8	Sum of packed sizes in bits of all fields in the fixed part and the largest variant, plus sum of sizes in bits of any holes created to meet alignment requirements; if sum > 32, sum is rounded up to the next multiple of 8	
VARYING OF CHAR	Maximum length + 2	(Maximum length + 2) * 8	

Table 1-4: (Cont.) Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits	
Unpacked SET <sup>3</sup>	32, if the set base type is subrange of INTEGER or UNSIGNED. Else, compute (ORD + 8) DIV 8. If this result < = 8, result is rounded up to 1, 2, 4, or 8.	Compute (ORD (upper-bound) + 1). If result < = 64, result is rounded up to 8, 16, 32, or 64. If result > 64, result is rounded to next higher multiple of 8.	
PACKED SET <sup>3</sup>	Result of (ORD(upperbound) + 8) DIV 8.	Compute (ORD (upper-bound) + 1). If result > 32, result is rounded to next higher multiple of 8.	
FILE	Not specified	Not specified	

<sup>&</sup>lt;sup>3</sup>Sets of type INTEGER and UNSIGNED are limited to 256 bits.

You can discover both the unpacked and packed sizes for variables of any type by using the predeclared functions BITNEXT, BITSIZE, NEXT, and SIZE. These functions, which are fully described in *Programming in VAX PASCAL*, require a parameter that is the name of either a type or a variable:

- BITNEXT returns an integer value that indicates what the packed size would be for an array component of the type.
- BITSIZE returns an integer value that indicates what the packed size would be for a record field of the type.
- NEXT returns an integer value that indicates what the unpacked size would be for an array component of the type.
- SIZE returns an integer value that indicates what the unpacked size would be for a variable or record field of the type.

The following examples illustrate the effects of packing records and multidimensional arrays at various levels.

### Examples

1. TYPE

Internal\_Arr = ARRAY[1..5] of 0..6;

VAR

Samp1\_Arr : PACKED ARRAY[1..5] OF Internal\_Arr;

Each component of an array of type Internal\_Arr is stored in a longword. Each component of Samp1\_Arr, in turn, requires five longwords—enough storage space for five components of type Internal\_Arr. The entire array Samp1\_Arr therefore occupies 25 longwords (800 bits).

VAR

Samp1\_Arr : ARRAY[1..5] OF PACKED ARRAY[1..5] OF O..6;

Each PACKED ARRAY[1..5] of 0..6 requires 15 bits. Because the packed arrays are components of an unpacked array, their size is rounded up to an even 16 bits. The total size of Samp1\_Arr is therefore 80 bits.

3. TYPE

Internal\_Arr = PACKED ARRAY[1..5] OF 0..6;

VAR

Samp2\_Arr : PACKED ARRAY[1..5] OF Internal\_Arr; Samp3\_Arr : PACKED ARRAY[1..5,1..5] OF 0..6;

In this example, every component of Internal\_Arr requires only three bits because the array is packed. Each component of Samp2\_Arr and Samp3\_Arr can be stored in 15 bits, and each array occupies 75 bits. Except when the program is compiled with the /OLD\_ VERSION qualifier (see Appendix E, Differences Between Version 1 and Subsequent Versions of *Programming in VAX PASCAL*), the specification of PACKED for an array with multiple indexes results in packing at every level. Therefore, the two arrays in this example are equivalent.

4. VAR

Sample : PACKED ARRAY[1..5,1..5,1..5] OF 0..6;

This example shows space savings for arrays of more than two dimensions when PACKED is specified at every level. The subrange 0..6 requires 3 bits; five 3-bit components require 15 bits. This size describes the innermost dimension of Sample. Next, five 15-bit components require 75 bits. Because of the 32-bit rule, each 75-bit component is rounded up to 80 bits. This size describes the middle and inner dimensions of Sample. Finally, five 80-bit components require 400 bits (50 bytes). The entire array Sample, then, requires 400 bits.

5. VAR

Sample\_Rec : PACKED RECORD

Field\_1 : BOOLEAN;
Field\_2 : INTEGER;
Field\_3 : DOUBLE;

END:

In this example, Field\_1 requires only 1 bit of storage. Field\_2 is 32 bits large, and starts immediately following Field\_1. Since Field\_3 is larger than 32 bits, it will start on the next byte boundary. The entire record Sample\_Rec, then, requires 104 bits.

## 1.4 Alignment Boundaries

The memory addressing boundary on which a variable or a component is aligned depends on the variable's or the component's allocation size. The alignment can be changed by using the ALIGNED and UNALIGNED attributes, as explained in *Programming in VAX PASCAL*. The following conditions determine boundary alignment:

- If the variable or component has an alignment attribute, the compiler uses the specified alignment. You cannot apply the UNALIGNED attribute to a variable or component whose allocation size is greater than 32 bits. A variable or component of this size must be aligned on a byte boundary.
- Variables declared without an alignment attribute are aligned by default on a byte boundary. The compiler may align such variables on a larger storage boundary if it can access them more efficiently by doing so.
- All components of an unpacked array or record variable are byte aligned within the array or record. The array or record itself, however, may be unaligned.
- The alignment of components of a packed array, packed record, or VARYING OF CHAR variable always follows the 32-bit rule (see Section 1.3).
- Dynamic variables allocated by the NEW procedure are always aligned on a quadword boundary.

Although you can save storage space by packing variables of structured types, you must be careful to pack at the proper level. Except for its alignment, a record field whose type is an unpacked array, set, or record occupies the same amount of space in a packed or an unpacked record variable. To pack such a field, you must explicitly declare its type to be packed.

When packing multidimensional arrays, you must specify packing at the innermost level to gain any significant space advantage. For example, there is no advantage in packing an array of an unpacked structured type—the unpacked components will still be aligned on byte boundaries, thereby leaving holes in the storage space. To gain storage space, you must specify a packed array of a packed structured type.

#### Examples

1. VAR

```
X : [STATIC. ALIGNED (3)] INTEGER: (* $LOCAL, QUADWORD ALIGNED *)
```

In this example, X is declared a STATIC variable which is aligned aligned on a QUADWORD boundary.

VAR

```
Page_Name : [PSECT(New_Section)] PACKED ARRAY [1..512] OF
                              (* New_Section, PAGE ALIGNED *)
   [BYTE] 0..255;
```

This example declares the page aligned variable Page\_Name which is to be allocated in New\_Section.

3. VAR

```
X : PACKED RECORD
                                     (* AUTOMATIC *)
           Field1 : BOOLEAN;
           Field2 : REAL;
           Field3 : BOOLEAN;
           Field4 : DOUBLE;
           END:
```

In this example, Field1 begins at bit position 0 and is 1 bit long. Field 2 begins at bit position 1 and is 32 bits long. Field3 begins at bit position 33 and is 1 bit long. However, because Field4 is greater than 32 bits long (DOUBLE requires 64 bits) Field4 must be byte aligned. For this reason, Field4 begins on bit position 40 and is 64 bits long. See Section 1.3 for the 32-bit rule.

4. VAR

X : RECORD

Field1 : [BIT(3)] 0..7;

(\* AUTOMATIC \*) Field2 : [UNALIGNED] INTEGER;

In this example, Field1 of record X is declared to begin on bit position 0 and is 3 bits long. Due to the use of the UNALIGNED attribute on Field2, Field2 begins on bit position 3 and is 32 bits long. Note that you can also receive the same behavior by packing record X.

Without using the UNALIGNED attribute, or without X being a PACKED record, however, Field2 would have started on bit position 8.

#### 1.5 **Representation of Varying Data**

A variable of type VARYING OF CHAR is stored as though it were a PASCAL record type of the format:

Length : [WORD] O..Maxlength;

Body : PACKED ARRAY[1..Maxlength] OF CHAR;

END:

Storage is allocated as one byte per character, with an initial field of two bytes to indicate the total length. A variable of type VARYING OF CHAR whose length is greater than or equal to three characters is allocated an exact number of bytes. Storage allocation for a variable of type VARYING OF CHAR whose maximum length is zero, one, or two characters follows the 32-bit rule in that the variable can have the UNALIGNED attribute (see Section 1.3).

Figure 1-1 illustrates the storage allocated for a variable of type VARYING[8] OF CHAR.

Figure 1–1: Storage of VARYING Data

15	0
L	LENGTH
[2]	[1]
[4]	[3]
[6]	[5]
[8]	[7]
79	64

ZK-1038-82

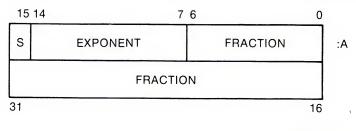
## 1.6 Representation of Floating-Point Data

The following sections summarize the internal representation of singleprecision (REAL and SINGLE types), double-precision (DOUBLE type), and quadruple-precision (QUADRUPLE type) floating-point numbers. For more detailed information, see the Introduction to VAX/VMS System Routines.

## **Single-Precision (SINGLE, REAL Types)**

A single-precision floating-point number is represented by four contiguous bytes. The bits are numbered from the right, 0 through 31, as shown in Figure 1–2.

Figure 1–2: Single-Precision Floating-Point Data Representation



ZK-1039-82

A single-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of an F\_floating-point value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 7 are an excess 128 binary exponent.
- Bits 6 through 0 and 31 through 16 are a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and from 0 through 6.

The 8-bit exponent field encodes the values 0 through 255:

- An exponent value of 0, with a sign bit of 0, indicates that the floatingpoint number has a value of 0.
- Exponent values of 1 through 255 indicate binary exponents of -127 through +127.
- An exponent value of 0, with a sign bit of 1, is considered a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

On the VAX, the value of a floating-point number is in the approximate range of .29 \*  $(10^{38})$  through 1.7 \*  $(10^{38})$ . The precision of a singleprecision value is approximately one part in  $2^{23}$ , or 7 decimal digits.

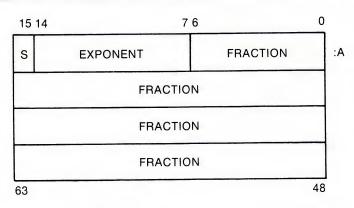
#### **Double-Precision (DOUBLE Type)** 1.6.2

A double-precision floating-point value is represented by eight contiguous bytes. Variables of type DOUBLE take one of two formats: D\_floating or G\_floating. D\_floating format allows values to be expressed with greater precision than that allowed by the G\_floating format; G\_floating format allows a wider range of values to be expressed than that allowed by the D\_floating format.

#### 1.6.2.1 **D\_Floating Point**

D\_floating-point data are stored in the format shown in Figure 1-3. The bits are numbered from the right, 0 through 63.

Figure 1-3: D\_Floating-Point Double-Precision Representation



ZK-1040-82

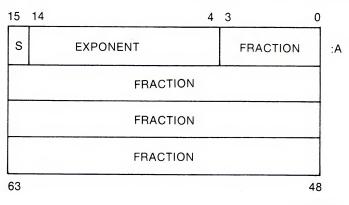
A D\_floating-point double-precision value is specified by its address A, the address of the byte containing bit 0. The form of a D\_floating-point value is identical to that of a single-precision floating-point value except for an additional 32 low-significance fraction bits. Within the fraction, bits of increasing significance are numbered 48 through 63, 32 through 47, 16 through 31, and 0 through 6.

The exponent conventions and approximate range of values are the same for D\_floating-point values as for single-precision floating-point values. The precision of a D\_floating-point value is approximately one part in 255, or 16 decimal digits.

### 1.6.2.2 G\_Floating-Point

G\_floating-point data are stored in the format shown in Figure 1–4. The bits are numbered from the right, 0 through 63.

Figure 1–4: G\_Floating-Point Double-Precision Representation



ZK-1041-82

A G\_floating-point double-precision value is specified by its address A, the address of the byte containing bit 0. The form of a G\_floating-point value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 4 are an excess 1024 binary exponent.
- Bits 3 through 0 and 63 through 16 represent a normalized 53-bit fraction without the redundant most significant fraction bit. Within the fraction, bits of increasing significance go from 48 through 63, 32 through 47, 16 through 31, and 0 through 3.

The 11-bit exponent field encodes the values 0 through 2047:

- An exponent value of 0, with a sign bit of 0, indicates that the G\_floating-point number has a value of 0.
- Exponent values of 1 through 2047 indicate binary exponents of -1023 through +1023.

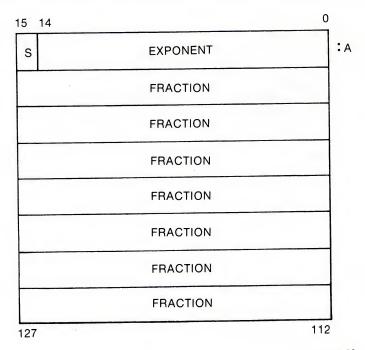
An exponent value of 0, together with a sign bit of 1, is considered a reserved operand. Floating-point instructions processing a reserved operand cause a reserved operand fault.

The value of a G\_floating-point number is in the approximate range of .56 \*  $(10^{-308})$  through .90 \*  $(10^{308})$ . The precision of a G\_floating-point value is approximately one part in  $2^{52}$ , or 15 decimal digits.

# 1.6.3 Quadruple-Precision (QUADRUPLE Type)

A quadruple-precision floating-point value is represented by 16 contiguous bytes. The bits are numbered from the right 0 through 127, as shown in Figure 1–5.

Figure 1-5: **Quadruple-Precision Floating-Point** Representation



ZK-1042-82

A quadruple-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of the value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 0 are an excess 16,384 binary exponent.
- Bits 127 through 16 are a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31.

The 15-bit exponent field encodes the values 0 through 32,767:

- An exponent value of 0, with a sign bit of 0, indicates that the quadruple-precision number has a value of 0.
- Exponent values of 1 through 32,767 indicate true binary exponents of -16,383 through +16,383.
- An exponent value of 0, with a sign bit of 1, is considered a reserved operand. Floating-point instructions processing a reserved operand cause a reserved operand fault.

The value of a quadruple-precision floating-point number is in the approximate range of  $0.84 * (10^{-4932})$  through  $0.59 * (10^{4932})$ . The precision of a quadruple-precision value is approximately one part in 2112, or 33 decimal digits.

# **Program Optimization and Efficiency**

The word optimization, as used in this chapter, refers to the process of improving the efficiency of programs. The objective of optimization is to produce source and object programs that achieve the greatest amount of processing with the least amount of time and memory.

Improved program efficiency results when programs are carefully designed and written, and when compilation techniques take advantage of machine architecture. The VAX PASCAL compiler produces efficient code by utilizing the benefits of the VAX native-mode architecture and hardware. The primary goal of optimization performed by the VAX PASCAL compiler is faster program execution.

The topics covered in this chapter are:

- Compiler optimizations
- Programming considerations
- Optimization considerations

# 2.1 Compiler Optimizations

Programs compiled with the VAX PASCAL compiler undergo, by default, the process known as optimization. Optimization refers to a set of techniques the compiler uses to minimize the amount of time and memory required to execute a program. An optimizing compiler automatically attempts to remove repetitious instructions and redundant computations by making assumptions about the values of certain variables. This in turn reduces the size of the object code, allowing a program written in a high-level language to execute at a speed comparable to that of a wellwritten assembly language program. Although optimization can increase

the amount of time required to compile a program, it results in a program that may execute faster and more efficiently than a nonoptimized program.

The language elements you use in the source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization (see Section 2.2). In addition, this awareness will often make it easier for you to track down the source of a problem when your program exhibits unexpected behavior.

The VAX PASCAL compiler performs the following optimizations:

- Compile-time evaluation of constant expressions
- Elimination of some common subexpressions
- Partial elimination of unreachable code
- Code hoisting from structured statements, including the removal of invariant computations from loops
- Inline code expansion for many predeclared functions
- Inline code expansion for user declared routines
- Rearranging unary minus and NOT operations to eliminate unary negation/complement operations
- Partial evaluation of logical expressions
- Propagation of compile-time known values

The preceding optimizations are described more thoroughly in the following sections. In addition, the VAX PASCAL compiler performs the following optimizations, which may be detected only by a careful examination of the machine code produced by the VAX PASCAL compiler.

- Global assignment of variables to registers—frequently referenced variables are assigned to registers (if possible) to reduce the number of memory references needed
- Reordering the evaluation of expressions to minimize the number of temporary values required
- Peephole optimization of instruction sequences—that is, examining code a few instructions at a time to find operations that can be replaced by shorter, faster equivalent operations

# **Compile-Time Evaluation of Constants**

The VAX PASCAL compiler performs the following computations on constant expressions at compile time.

Negation of constants—the value of a constant preceded by unary minus signs is negated at compile time. For example:

```
X := -10.0;
```

is compiled as a single move instruction.

Type conversion of constants—the value of a lower-ranked constant is converted to its equivalent in the data type of the higher-ranked operand at compile time. For example, if X and Y are both real numbers, then the operation

```
X := 10 * Y;
is compiled as
X := 10.0 * Y;
```

Arithmetic on integer and real constants—an expression that involves +, -, \*, or / operators is evaluated at compile time. For example:

```
CONST
   NN = 27;
I := 2 * NN + J;
is compiled as
I := 54 + J:
```

Array address calculations involving constant indexes—these are simplified at compile time whenever possible. For example:

```
I : ARRAY[1..10,1..10] OF INTEGER;
I[1,2] := I[4,5];
```

is compiled as a single move instruction.

Evaluation of constant functions and operators—arithmetic, ordinal, transfer, unsigned, allocation size, CARD, EXPO, and ODD functions involving constants, concatenation of string constants, and logical and relational operations on constants, are evaluated at compile time.

#### 2.1.2 **Elimination of Common Subexpressions**

The same subexpression often appears in more than one computation within a program. For example:

```
A := B * C + E * F:
H := A + G - B * C;
IF ((B * C) - H) <> 0
THEN
```

In this code sequence, the subexpression B \* C appears three times. If the values of the operands B and C do not change between computations, the value B \* C can be computed once and the result can be used in place of the subexpression. Thus, the sequence shown above is compiled as follows:

```
t := B * C;
A := t + E * F;
H := A + G - t;
IF ((t) - H) <> 0
THEN
```

As you can see, two computations of B \* C have been eliminated.

In this case, you could have modified the source program itself for greater program optimization. The following example shows a more significant application of this kind of compiler optimization, in which you could not reasonably modify the source code to achieve the same effect. Without optimization, the source program

```
VAR
  A, B : ARRAY[1..25,1..25] OF REAL;
A[I,J] := B[I,J];
would be compiled as
t1 := (J - 1) * 25 + I;
t2 := (J - 1) * 25 + I;
A[t1] := B[t2];
```

Variables t1 and t2 represent equivalent expressions. The VAX PASCAL compiler eliminates this redundancy by producing the following optimization:

```
t = (J - 1) * 25 + I;
A[t] := B[t];
```

#### 2.1.3 **Elimination of Unreachable Code**

The VAX PASCAL compiler can determine which lines of code (if any) will never be executed and eliminates that code from the object module being produced. For example, consider the following lines from a program:

```
CONST
   Debug_Switch = FALSE;
IF Debug_Switch
   WRITELN ('Error found here');
```

The IF-THEN statement is designed to write an error message if the value of the symbolic constant Debug\_Switch is TRUE. Suppose that the error has been removed, and you change the definition of Debug\_Switch to give it the value FALSE. When the program is recompiled, the compiler can determine that the THEN clause will never be executed because the IF condition is always FALSE; no machine code is generated for this clause. You need not remove the IF-THEN statement from the source program.

Note that code which is otherwise unreachable, but contains a label(s), is not eliminated unless the GOTO statement and the label itself are located in the same block.

#### 2.1.4 **Code Hoisting from Structured Statements**

The VAX PASCAL compiler can improve the execution speed and size of programs by removing invariant computations from structured statements. For example, if the compiler detected the following IF-THEN statement

```
FOR J :=1 TO I+23 DO
   BEGIN
   IF Selector
   THEN
      A[I + 23, J - 14] := 0
      B[I + 23, J - 14] := 1;
END:
```

it would recognize that, regardless of the Boolean value of Selector, a value is to be stored in the array component denoted by [I + 23, J - 14]. Thus, the compiler would change the sequence to:

```
t := I + 23:
FOR J :=1 TO t DO
BEGIN
  u := J - 14;
  IF Selector
     A[t,u] := 0
  ELSE
     B[t,u] := 1;
END:
```

This removes the calculation of (J - 14) from the IF statement, and the calculation of (I + 23) from both the IF statement and the loop.

# 2.1.5 Inline Code Expansion for Predeclared Functions

The VAX PASCAL compiler can often replace calls to predeclared routines with the actual algorithm for performing the calculation. For example, the compiler replaces the function call

```
Square := SQR (A);
with
Square := A * A;
```

and generates machine code based on the expanded call. The program will execute faster because the algorithm for the SQR function has already been included in the machine code.

#### **Inline Code Expansion for User Declared Routines** 2.1.6

Inline code expansion for user declared routines performs in the same manner as inline code expansion for predeclared functions—the VAX PASCAL compiler can often replace calls to user-declared routines with an inline expansion of the routine's executable code. Inline code expansion is useful, for example, on routines that are called only a few times. The overhead of an actual procedure call is avoided; therefore, program execution is faster. The size of the program, however, may increase due to the routine's expansion. There are several restrictions on inline code expansion:

- 1. The called routine may not be an EXTERNAL routine.
- 2. The calling routine must have inline optimization enabled.
- 3. Both the calling and called routines must have the same checking options enabled.
- 4. Both the calling and called routines must use the same program section.
- 5. The called routine may not declare a routine parameter or itself be a routine parameter.
- 6. The called routine must not establish an exception handler, nor can it be used as an exception handler.
- 7. The called routine's parameter list may not use either the LIST or TRUNCATE attributes, or contain a READONLY VAR parameter.
- 8. The called routine may not declare file variables, labels, or contain non-local GOTO's.

9. The called routine must not return a value of a structured type.

#### NOTE

There is no stack frame for an inline user declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline routine is therefore difficult, and is not recommended.

# 2.1.7 Rearranging Operations

The VAX PASCAL compiler can produce more efficient machine code by rearranging operations to avoid having to negate and then calculate the complement of the values involved. For example, if a program includes an operation such as

```
(-C) * (B - A)
```

the compiler rearranges the operation to read

```
C * (A - B)
```

These two operations produce the same result, but, because the compiler has eliminated negation/complement operations, the machine code produced will be more efficient.

# 2.1.8 Partial Evaluation of Logical Expressions

The PASCAL language does not specify the order in which the components of an expression must be evaluated. If the value of an expression can be determined by partial evaluation, then some subexpressions may not be evaluated at all. This situation occurs most frequently in the evaluation of logical expressions. For example, in the following WHILE statement

```
WHILE (I < 10) AND (A[I] <> 0) DO

BEGIN

A[I] := A[I] + 1;

I := I + 1;

END;
```

the order in which the two subexpressions (I < 10) and (A[I] < > 0) are evaluated is not specified; in fact, the compiler may evaluate them simultaneously. Regardless of which subexpression is evaluated first, if its value is FALSE, the condition being tested in the WHILE statement is also FALSE. The other subexpression need not be evaluated at all. Thus, in this case, the body of the loop is never executed.

# 2.1.9 Value Propagation

The compiler keeps track of the values assigned to variables and traces the values to most of the places that they are used. If it is more efficient to use the value rather than a reference to the variable, the compiler makes this change. This optimization is called value propagation. Value propagation causes the object code to be smaller, and may also improve run-time speed.

Value propagation performs the following two actions:

It allows run-time operations to be replaced with compile-time operations. For example, in a program that includes the following assignments:

```
PI := 3.14;
PIOVER2 := PI/2;
```

The compiler will recognize the fact that PI's value did not change between the time of PI's assignment and its use. Thus, the compiler would use PI's value instead of a reference to PI and perform the division at compile time. The compiler would treat the assignments as if they were:

```
PI := 3.14;
PIOVER2 := 1.57;
```

Note that this process is repeated, allowing for further constant propagation to occur.

It allows comparisons and branches to be avoided at run-time. For example, in a program which included the following operations:

```
X := 3:
IF X <> 3
THEN
  Y := 30
ELSE
   Y := 20;
```

The compiler would recognize that X was 3 and the THEN statement could not be reached. Thus the compiler would generate code as if the statements were written:

X := 3; Y := 20;

#### 2.1.10 **Alignment of Compiler-Generated Labels**

The VAX PASCAL compiler aligns the labels it generates for the top of loops and the beginnings of ELSE branches on longword boundaries, filling in unused bytes with NOP instructions. On the VAX, a branch to a longword-aligned address is faster than a branch to an unaligned address. This optimization may increase the size of the generated code, however, it increases run-time speed.

#### 2.1.11 **Reducing Errors Through Optimization**

An optimized program produces results and run-time diagnostic messages identical to those produced by an equivalent unoptimized program. An optimized program may produce fewer run-time diagnostics, however, and the diagnostics may occur at different statements in the source program. For example:

Unoptimized Code	Optimized Code		
A := X/Y;	t := X/Y;		
B := X/Y;	A := t;		
FOR I := 1 TO 10 DO	B := t;		
C[I] := C[I] * X/Y;	FOR I := 1 TO 10 DO		
	C[I] := C[I] * t;		

If the value of Y is 0.0, the unoptimized program produces 12 divideby-zero errors at run-time; the optimized program produces only one. (Note that t is a temporary variable created by the compiler.) Eliminating redundant calculations and removing invariant calculations from loops can affect the detection of such arithmetic errors. You should keep this point in mind when you include error-detection routines in your program.

# 2.2 Programming Considerations

The VAX PASCAL language elements that you use in a source program directly affect the compiler's ability to optimize the resulting object program. Therefore, you should be aware of the following ways in which you can assist compiler optimization and thus obtain a more efficient program.

- Define constant identifiers to represent values that do not change during your program. The use of constant identifiers generally makes a program easier to read, understand, and modify later. In addition, the resulting object code is more efficient because symbolic constants are evaluated only once, at compile time, while variables must be reevaluated whenever they are assigned new values.
- Whenever possible, use the structured control statements CASE, FOR, IF-THEN, IF-THEN-ELSE, REPEAT, WHILE, and WITH rather than the GOTO statement. Although the GOTO statement can be used to exit from a loop, careless use of it interferes with both optimization and the straightforward analysis of program flow.

#### NOTE

When both the REPEAT and WHILE statements are valid in a loop you should use the REPEAT statement because it allows for faster execution.

Enclose in parentheses any subexpression that occurs frequently in your program. The compiler checks whether any assignments have affected the subexpression's value since its last occurrence. If the value has not changed, the compiler recognizes that a subexpression enclosed in parentheses has already been evaluated and does not repeat the evaluation. For example:

```
X := SIN (U + (B - C));
Y := COS (V + (B - C));
```

The compiler evaluates the subexpression (B - C) as a result of performing the SIN function. When it is encountered again, the compiler checks to see whether new values have been assigned to either B or C since they were last used. If their values have not changed, the compiler does not reevaluate (B - C).

Once your program has been completely debugged, disable all checking with either the CHECK(NONE) attribute or the /NOCHECK or /CHECK=NONE compile-time qualifier (see Programming in VAX PASCAL). When no checking code is generated, more optimizations can occur, and the program will execute faster.

Note that integer overflow checking is disabled by default. If you are sure that your program is not in danger of integer overflow, you should not enable overflow checking. Because overflow checking precludes certain optimizations, you can achieve a more efficient program by leaving it disabled.

- When a variable is accessed by a program block other than the one in which it was declared, the variable should have static rather than automatic allocation. A variable is statically allocated if it is declared at program or module level or if it is associated with either the STATIC or PSECT attribute in a nested block. A static variable has a fixed location in memory and is therefore easy to access. An automatically allocated variable, on the other hand, has a varying location in memory; accessing it in another block is time-consuming and less efficient.
- Avoid using the same temporary variable many times in the course of a program. Instead, use a new variable every time your program needs a temporary variable. Because variables stored in registers are the easiest to access, your program is most efficient when as many variables as possible can be allocated in registers. If you use several different temporary variables, the lifetime of each one is greatly reduced; thus, there is a greater chance that storage for them can be allocated in registers rather than at memory locations.

# 2.3 Optimization Considerations

Because the compiler must make certain assumptions in order to optimize a program, unexpected results may occur if you do not utilize the VAX PASCAL optimizations discussed in the following sections. If your program does not execute correctly because of undesired optimizations, you can use the NOOPTIMIZE attribute or the /NOOPTIMIZE compile-time qualifier (see Programming in VAX PASCAL) to prevent optimizations from occurring.

#### **Subexpression Evaluation** 2.3.1

An optimizing compiler, as discussed in Section 2.1, can evaluate subexpressions in any order and may even choose not to evaluate some of them. These considerations are important when the subexpressions designate functions that have side effects. For example, the following IF statement

IF F(A) AND F(B) THEN

contains two designators for function F. If F has side effects, the compiler does not guarantee the order in which the side effects will be produced. In fact, if one call to F returns FALSE, the other call to F might never be executed, and the side effects that result from that call would never be produced.

Similarly, the following expression

Q := F(A) + F(A);

consists of two designators for function F with the same parameter A. The PASCAL standard allows a compiler to optimize the code so that

Q := 2 \* F(A)

If the compiler does so, and function F has side effects, the side effects would occur only once because the compiler has generated code that evaluates F(A) only once.

If you disable optimization while your program is being compiled, the VAX PASCAL compiler will evaluate all expressions completely and from left to right.

### 2.3.2 Lowest Negative Integer

The compiler assumes that all integer values are in the range –MAXINT through MAXINT. However, the VAX architecture supports an additional integer value, (–MAXINT–1). Should your program contain a subexpression with this value, its evaluation might result in an integer overflow trap. Therefore, a computation involving the value (–MAXINT–1) might not produce the expected result. If you want to evaluate expressions that include (–MAXINT–1), you should disable either optimization or integer overflow checking.

### 2.3.3 Pointer References

The compiler assumes that the value of a pointer variable is either the constant identifier NIL or a reference to a variable allocated in heap storage by the NEW procedure. A variable allocated in heap storage is not declared in a VAR section and has no identifier of its own; you can refer to it only by the name of a pointer variable, followed by a circumflex (^).

If a pointer variable in your program must refer to a variable with an explicit name, that variable must be declared VOLATILE or READONLY. The compiler makes no assumptions about the value of VOLATILE variables and therefore performs no optimizations on them (the VOLATILE attribute is fully described in *Programming in VAX PASCAL*).

Use of the ADDRESS function, which creates a pointer to a variable, can result in a warning message because of optimization characteristics. By passing a nonreadonly or nonvolatile static or automatic variable as the parameter to the ADDRESS function, you indicate to the compiler that the variable was not allocated by NEW, but was declared with

its own identifier. Since this declaration violates the compiler's assumptions, a warning message occurs (the ADDRESS function is fully described in Programming in VAX PASCAL). Note that you may also use IADDRESS, which functions similarly to the ADDRESS function, except that IADDRESS does not generate any warning messages. You should use caution when using IADDRESS—see Programming in VAX PASCAL for more information.

Similarly, when the parameter to ADDRESS is a formal VAR parameter or a component of a formal VAR parameter, the compiler cannot be sure that only dynamic variables allocated by NEW will be passed to the function. Thus, a warning message also results in this situation.

### 2.3.4 Variant Records

Because all the variants of a record variable are stored in the same memory location, a program can use several different field identifiers to refer to the same storage space. However, only one variant is valid at a given time; all other variants are undefined. Thus, you must store a value in a field of a particular variant before you attempt to use it. For example:

```
VAR
  X : INTEGER;
   A : RECORD
        CASE T : BOOLEAN OF
        TRUE : (B : INTEGER) :
        FALSE : (C : REAL);
        END:
X := A.B + 5;
A.C := 3.0;
X := A.B + 5;
```

Record A has two variants, B and C, which are located at the same storage address. When the assignment A.C := 3.0 is executed, the value of A.B becomes undefined because TRUE is no longer the currently valid variant. When the statement X := A.B + 5 is executed for the second time, the value of A.B is unknown. The compiler may choose not to evaluate A.B a second time because it has retained the field's previous value. To eliminate any misinterpretations caused by this assumption, variable A should be associated with the VOLATILE attribute. The compiler makes no assumptions about the value of VOLATILE objects (see *Programming in VAX PASCAL*).

#### 2.3.5 Type Cast Operations

When a type cast operation is performed, the compiler disregards any previous assumptions it had about the value of the cast object. This allows you to temporarily alter the type of the cast object at that point only. A type cast operation affects optimization only at the location in the program where the cast takes place. Optimizations elsewhere in the program and optimizations involving only uncast objects are not affected.

You should use type casts with care since the type cast operation can sometimes affect distant parts of a program. If a type cast on a variable is likely to affect its value at other points in the program, the variable should be declared VOLATILE (see *Programming in VAX PASCAL*).

# 2.3.6 Effects of Optimization on Debugging

When you compile a PASCAL program, the resulting object code is optimized by default. The compiler automatically allocates variables in registers, removes invariant expressions that occur within loops so that they are evaluated outside the loop, and so on.

Some of the effects of optimized programs on debugging are as follows:

Use of registers. When the VAX PASCAL compiler determines that the value of an expression does not change between two given occurrences, it may save the value in a register. In such a case, it does not recompute the value for the next occurrence, but simply assumes that the value saved in the register is valid. If, while debugging the program, you use the DEPOSIT command to change the value of the variable in the expression, then the value of that variable is changed, but the corresponding value stored in the register is not. Thus, when execution continues, the value in the register may be used instead of the changed value in the expression, causing unexpected results.

When the value of a variable is being held in a register, its value in memory is generally invalid, therefore, a spurious value may be displayed if the EXAMINE command is issued for a variable under these circumstances.

- Coding order. Some of the compiler optimizations cause code to be generated in a different order than it appears in the source. Sometimes code is eliminated altogether. This causes unexpected behavior when stepping by line or using the source display features of DEBUG.
- Use of condition codes. This optimization technique takes advantage of the way in which the VAX processor condition codes are set. For example, consider the following source code:

```
X := X + 2.5;
IF X < 0
THEN
```

Rather than test the new value of X to determine whether to branch, the optimized object code bases its decision on the condition code settings after 2.5 is added to X. Thus, if you attempt to set a debugging breakpoint at the second line and deposit a different value into X, you will not achieve the intended result because the condition codes no longer reflect the value of X. In other words, the decision to branch is being made without regard to the deposited value of the variable.

Inline code expansion on user declared routines. There is no stack frame for an inline user declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline user declared routine is therefore difficult, and is not recommended.

To prevent conflicts between optimization and debugging, you should always compile your program with the /NOOPTIMIZE qualifier until it is thoroughly debugged. Then you can recompile the program (which by default will be optimized) to produce efficient code.



# **Calling Conventions**

This chapter describes how to call routines that are not written in PASCAL and provides information on calling VAX/VMS system services and VAX Run-Time Library procedures. See *Programming in VAX PASCAL* for information on declaring and calling PASCAL routines.

The VAX/VMS Run-Time Library Routines Reference Manual and the Introduction to VAX/VMS System Routines both contain detailed information about procedure-calling and argument-passing mechanisms. You should be familiar with these subjects before you attempt to use the features described here. Note that the manuals referred to use the term "procedure" to mean any routine entered by a CALL instruction. This chapter uses the term "routine" instead, to avoid confusion with PASCAL's definition of "procedure."

# 3.1 VAX Procedure Calling Standard

Programs compiled by the VAX PASCAL compiler conform to the standard defined for VAX procedure calls (see the Introduction to VAX/VMS System Routines). This standard describes how parameters are passed, how function values are returned, and how routines receive and return control. By means of the calling standard, VAX PASCAL provides features that allow programs to call system services and routines written in other native-mode languages supported by VAX/VMS.

### 3.1.1 Parameter Lists

Each time a routine is called, the VAX PASCAL compiler constructs a parameter list. The VAX Procedure Calling Standard defines a parameter list as a sequence of longword (4-byte) entries. The first byte of the first entry in the list is a parameter count, which indicates how many parameters follow in the list.

The form in which the parameters in the list are represented is determined by the passing mechanisms you specify in the formal parameter list and the values you pass in the actual parameter list. The parameter list contains the actual parameters passed to the routine.

### 3.1.2 Function Return Values

In PASCAL, a function returns to the calling block the value that was assigned to its identifier during execution. The compiler chooses one of three methods for returning this value; the method chosen depends on the amount of storage required for values of the type:

- If the value can be represented in 32 bits of storage, it is returned in register R0.
- If the value requires from 33 to 64 bits, the low-order bits of the result are returned in register R0 and the high-order bits are returned in register R1.
- If the value is too large to be represented in 64 bits or if its type is a string type (PACKED ARRAY OF CHAR or VARYING OF CHAR), the calling routine allocates the required storage. An extra parameter—a pointer to the location where the function result will be stored—is added to the beginning of the calling routine's actual parameter list.

For values of structured types, the amount of storage required for the entire structure determines which of the three return methods is used. Character strings are always returned by the extra-parameter method.

Note that functions that require the use of an extra parameter can have no more than 254 parameters; functions that store their results in registers R0 and R1 can have 255 parameters.

Table 3–1 lists the methods by which values of each type are returned.

Table 3-1: **Function Return Methods** 

Type	Return Method		
INTEGER, UNSIGNED, CHAR, BOOLEAN, REAL, SINGLE, Pointer, Enumerated, Subrange	General Register R0		
DOUBLE	R0: Low-order result R1: High-order result		
QUADRUPLE, VARYING OF CHAR, PACKED ARRAY OF CHAR	Extra-parameter		
Other structured types	Depends on amount of storage required		

#### 3.1.3 **Contents of the Call Stack**

A call stack is a temporary area of storage allocated by the system for each user process. On the call stack, the system maintains information about each routine call in the current image. Each time a routine is called by a PASCAL program, the hardware creates a structure on the call stack; this structure is known as the call frame. The call frame for each active routine contains:

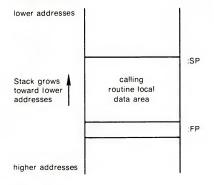
- A pointer to the call frame of the previous routine call. This pointer is called the saved Frame Pointer (FP).
- The saved Argument Pointer (AP) of the previous routine call.
- The storage address of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the saved Program Counter (PC).
- The saved contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

When execution of a routine ceases, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

The VAX PASCAL compiler uses the VAX CALLS and CALLG instructions to call routines. Figure 3-1 illustrates the events that occur during a routine call and shows the structure of the call stack after each event.

## Figure 3-1: Contents of the Run-Time Stack

### Before routine call:



Stack Pointer Frame Pointer

#### The calling routine's actions: First: Decrements SP by 4 times number of parameters & stores actual parameters parameter 1 called routine parameters parameter n calling routine local data area Second: :FP Calculates "static link" to allow the called routine to find the stack frame of its declaring routine. Stores static link in R1.

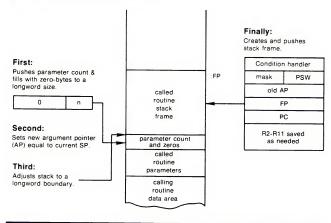
ZK-1037/1-82

Finally:

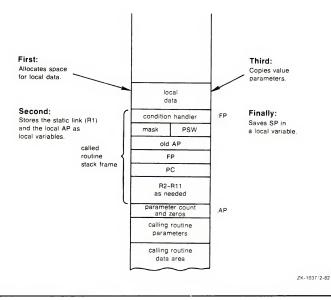
Issues CALLS instruction.

# Figure 3-1 (Cont.): Contents of the Run-Time Stack

#### 3 The CALLS instruction's actions:



#### 4 The called routine's actions:



As shown in Figure 3–1, the frame pointer of the calling routine is stored in R1, thus creating a link between the calling routine and the called routine. Because this link exists, the called routine can access variables and routines declared in enclosing blocks and can use GOTO statements to access executable statements in enclosing blocks.

However, if you declare a routine with the UNBOUND attribute, the system does not assume that the frame pointer of the declaring routine is stored in R1; thus, no link is established. As a result, an UNBOUND routine has the following restrictions:

- It may not access automatic variables declared in enclosing blocks.
- It may not call bound routines declared in enclosing blocks.
- It may not use a GOTO statement to transfer control to enclosing blocks other than the main program block.

By default, routines declared at program or module level and all other routines declared with the INITIALIZE, GLOBAL, or EXTERNAL attributes have the characteristics of UNBOUND routines. Routines passed by the immediate value mechanism (see Section 3.2.2) must be UNBOUND.

Asynchronous system trap routines (AST's) and RMS completion routines must have both the ASYNCHRONOUS and UNBOUND attributes. Because they are ASYNCHRONOUS, such routines may access only VOLATILE variables, predeclared routines, and other ASYNCHRONOUS routines. Note that the VAX PASCAL run-time system does not permit a program and an ASYNCHRONOUS routine (such as an AST) to access the same file simultaneously. See Programming in VAX PASCAL for more information on the ASYNCHRONOUS and UNBOUND attributes.

# 3.2 Passing Parameters to External Routines

Non-PASCAL routines require parameters in the form of addresses, immediate values, or descriptors. Following the VAX Procedure Calling Standard, VAX PASCAL defines three mechanisms by which parameters in such forms can be passed to routines:

- By reference—the parameter is the address of the value
- By immediate value—the parameter is the value itself
- By descriptor—the parameter is the address of a descriptor of the value

To indicate which mechanism is being used to pass parameters to external routines, VAX PASCAL provides the mechanism specifiers %REF, %IMMED, %DESCR, and %STDESCR. In addition, you can invoke the by-reference mechanism by indicating value and variable semantics in the declarations of formal parameters to non-PASCAL routines. Table 3-2 illustrates the types of parameters that can be passed to foreign mechanism parameters.

Table 3-2: **Foreign Mechanism Parameters** 

Parameter		Mechanism		
Туре	%IMMED	%REF	%DESCR	%STDESCR
Ordinal	Yes	Yes	Yes	No
SINGLE	Yes	Yes	Yes	No
DOUBLE or QUADRUPLE	No	Yes	Yes	No
RECORD	Yes <sup>1</sup>	Yes	No	No
ARRAY	Yes <sup>1</sup>	Yes	Yes	Yes <sup>2</sup>
ARRAY OF VARYING OF CHAR	No	Yes	Yes	No
Conformant ARRAY	Yes <sup>1</sup>	Yes	Yes	Yes <sup>2</sup>
Conformant ARRAY OF VARYING OF CHAR <sup>3</sup>	No	Yes	Yes	No
VARYING OF CHAR	No	Yes	Yes	No
Conformant VARYING OF CHAR	No	Yes	Yes	No
SET	Yes <sup>1</sup>	Yes	Yes	No
FILE	No	Yes	No	No
Pointer	Yes	Yes	Yes	No
PROCEDURE or FUNCTION	Yes <sup>4</sup>	Yes	Yes	No

<sup>&</sup>lt;sup>1</sup>Must follow the 32-bit rule.

<sup>&</sup>lt;sup>2</sup>Only if PACKED ARRAY OF CHAR.

<sup>&</sup>lt;sup>3</sup>Component type can be a conformant VARYING schema.

<sup>&</sup>lt;sup>4</sup>Must be UNBOUND.

A formal parameter is the argument(s) located in the heading of a procedure or function. An actual parameter is the argument(s) that corresponds to the formal parameter; it is specified in the call to the routine. In the following example

```
PROCEDURE SQRT (X : INTEGER);
.
.
BEGIN
SQRT (52)
END;
```

the variable X in the procedure heading is the formal parameter, and the argument (52) is the actual parameter.

A mechanism specifier usually appears before the name of a formal parameter, or in the attribute list of the formal parameter. However, in VAX PASCAL, a mechanism specifier can also appear before the name of an actual parameter. In the latter case, the specifier overrides the type, semantics, and mechanism specified in the formal parameter declaration. See Section 3.2.4 for more information.

The following sections describe how to specify the correct mechanism for passing parameters from VAX PASCAL programs to non-PASCAL routines. For information about passing parameters to VAX PASCAL routines from non-PASCAL routines, see Section 3.3. For a description of parameter passing between PASCAL routines, see *Programming in VAX PASCAL*.

# 3.2.1 By-Reference Mechanism

The by-reference mechanism passes the address of an actual parameter. By default, the VAX PASCAL compiler uses this mechanism to pass all actual parameters except those that correspond to conformant parameters.

The by-reference mechanism has three interpretations: value semantics, variable semantics, and foreign semantics. While the by-reference mechanism indicates how parameters are passed from a PASCAL routine to an external one, the semantics indicates how the called routine manipulates and returns parameters.

You can invoke the by-reference mechanism in four ways:

- By including neither the reserved word VAR nor a mechanism specifier before a formal parameter name. This syntax implies PASCAL value semantics, which is the default (see Section 3.2.1.1).
- By preceding a formal parameter name with the reserved word VAR. This syntax implies PASCAL variable semantics (see Section 3.2.1.2).
- By preceding a formal parameter name by the %REF mechanism specifier in the formal parameter's attribute list. Depending on the context, %REF can imply either variable or foreign semantics (see Section 3.2.1.3).
- By preceding a formal parameter name with the [REFERENCE] attribute in the formal parameter's attribute list. Depending on the context, [REFERENCE] can imply either variable or foreign semantics (see Section 3.2.1.3).

#### 3.2.1.1 Value Semantics

When you specify value semantics, the address of the actual parameter is passed to the called routine, which then copies the value from the specified address to local storage. You should use this method only when the called routine does not change the value of the actual parameter.

When you use value semantics, the actual parameter can be a compiletime or run-time expression; the names of routines are not allowed as value parameters.

For example, the following function declaration requires an actual parameter to be passed by reference using value semantics:

```
[EXTERNAL] FUNCTION MTH$TANH
(Angle : REAL)
: REAL;
EXTERN;
```

Hyperbolic\_Tan := MTH\$TANH (Radians);

The function MTH\$TANH returns, as a real value, the hyperbolic tangent of its parameter. The input parameter to this function is the size of the angle in radians. This value, which is not changed by MTH\$TANH, is passed using value semantics.

#### 3.2.1.2 Variable Semantics

When you specify variable semantics, the address of the actual parameter is passed to the called routine. In contrast to value semantics, however, the called routine directly accesses the actual parameter. Thus, execution of the called routine can change the value of the actual parameter.

When you use variable semantics, the actual parameter must be a variable or a component of an unpacked structured variable; no other expressions are allowed unless the formal parameter has the READONLY attribute (see *Programming in VAX PASCAL*). The names of routines are never allowed as variable parameters.

You must use variable semantics when passing a file variable as an actual parameter. This method is also useful for preventing the copying of large parameters.

The following function declaration requires an actual parameter to be passed by reference using variable semantics:

```
TYPE

$QUAD = [QUAD,UNSAFE] RECORD

LO: UNSIGNED;
L1: INTEGER;
END;

[ASYNCHRONOUS,EXTERNAL(SYS$GETTIM)] FUNCTION $GETTIM

(VAR TIMADR: [VOLATILE] $QUAD)

: INTEGER;
EXTERNAL;

VAR

Systime: $QUAD;
...
Return := $GETTIM (Systime);
```

This example declares an external routine, SYS\$GETTIM, which returns the system time. The input parameter is a 64-bit variable into which the system service writes the time. Because the function changes the value of the actual parameter Systime, Systime is passed by variable semantics.

### 3.2.1.3 %REF Mechanism Specifier

When you use the %REF mechanism specifier, the address of the actual parameter is passed to the called routine, which is then allowed to change the value of the corresponding actual parameter. The semantics used depends on the data item represented by the actual parameter:

- If the actual parameter is a variable, %REF on the actual or formal parameter implies variable semantics within the called routine.
- If the actual parameter is an expression or a variable enclosed in parentheses, %REF implies foreign semantics.

Under the rules of foreign semantics, the calling routine makes a copy of the value and passes the address of the copy to the called routine. Although the called routine is allowed to change this value, no change is reflected when control returns to the calling routine. Note how foreign semantics differ from value semantics: with foreign semantics, the copy is made by the *calling* (PASCAL) routine, whereas with value semantics, the copy is made by the *called* (external) routine.

 If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY.

The following procedure declaration requires two parameters passed by reference:

```
TYPE
   Matrix : ARRAY[10,10] OF REAL;

VAR
   A, B, Q : Matrix;

PROCEDURE Find_Determinant
   ([REFERENCE] Det : Matrix;
   VAR Inv : Matrix);
```

The procedure Find\_Determinant performs a matrix inversion and calculates the determinant. The following are legal calls to Find\_Determinant:

```
Find_Determinant (A, Q);
Find_Determinant ((A), Q);
```

In the first call, A is passed by reference using variable semantics. In the second call, the calling routine copies the value of A and passes it by reference to Find\_Determinant. In both calls, Q is passed by reference using variable semantics.

### 3.2.1.4 [REFERENCE] Attribute

The [REFERENCE] attribute has the same properties as the %REF mechanism specifier; see Section 3.2.1.3 for more information.

# 3.2.2 By-Immediate-Value Mechanism

The by-immediate mechanism passes a copy of a value instead of passing the address.

The by-immediate mechanism has two interpretations: the %IMMED mechanism specifier, and the [IMMEDIATE] attribute.

### 3.2.2.1 %IMMED Mechanism Specifier

When you use the %IMMED mechanism specifier, the compiler passes a copy of a value rather than an address. Only formal value and routine parameters declared in external routines can have the %IMMED specifier. The actual parameter can be a compile-time or run-time expression, or it can be a routine identifier. If the actual parameter expression is enclosed in parentheses, the %IMMED specifier applies to a copy of the expression's value or of the function result.

Variables that require more than 32 bits of storage, including file variables, cannot be passed by immediate value.

The following function declaration requires an actual parameter to be passed by immediate value:

```
[ASYNCHRONOUS,EXTERNAL(SYS$WAITFR)] FUNCTION $WAITFR
   (%IMMED EFN : INTEGER)
   : INTEGER;
   EXTERNAL;

VAR
   Event_Flag : INTEGER;
```

Status := \$WAITFR (Event\_Flag);

The external routine SYS\$WAITFR, which waits for a specific event flag, requires one input parameter: the number of the event flag for which to wait. This number is passed as an immediate value, copied from the integer variable Event\_Flag.

Normally, an actual procedure or function parameter is passed as the address of a bound procedure value. The bound procedure value's data type, which is specific to VAX, consists of two longword entries. The first entry is the address of the entry mask. The second entry is the routine's static scope pointer, the address of the call frame in which the routine was declared. However, when a routine is passed by immediate value, the argument list contains only one entry, the address of the entry mask. No static scope pointer is passed; therefore, any routine passed to an immediate parameter should access only its local variables and those variables with static allocation. For more information on bound procedure values, see the *Introduction to VAX/VMS System Routines*.

The following procedure declaration requires a procedure parameter to be passed by immediate value:

[EXTERNAL] PROCEDURE Forcaller
 ([IMMEDIATE, UNBOUND] PROCEDURE Utility);
 FORTRAN;

The PASCAL procedure Utility is passed by immediate value to the FORTRAN subroutine Forcaller. The argument list contains the address of Utility's entry mask, which indicates the registers to be saved.

### 3.2.2.2 [IMMEDIATE] Attribute

The [IMMEDIATE] attribute has the same properties as the %IMMED mechanism specifier; see Section 3.2.2 for more information.

# 3.2.3 By-Descriptor Mechanism

When you use the by-descriptor mechanism, the compiler passes the address of a string, array, or scalar descriptor, as described in the *Introduction to VAX/VMS System Routines*. A descriptor is a data structure that contains the address of a parameter and other information, such as the parameter's data type and size. The VAX PASCAL compiler generates the descriptor and supplies the necessary information.

VAX PASCAL includes the %STDESCR mechanism specifier for passing fixed-length string descriptors and the %DESCR mechanism specifier for passing array, VARYING string, and scalar descriptors. You cannot pass a component of a packed structure using either of these specifiers unless you enclose the component in parentheses. Note that an array descriptor is passed by default to a formal conformant array parameter,

and a varying-string descriptor is passed by default to a formal conformant varying parameter.

Table 3-3 lists the class and type of descriptor generated for parameters that can be passed using the by-descriptor mechanism. See the Introduction to VAX/VMS System Routines for complete information on descriptor classes and types.

Table 3-3: Parameter Descriptors

Parameter Type		Descriptor Class and Type	
	%DESCR	%STDESCR	Value or VAR Semantic
Ordinal	DSC\$K_CLASS_S <sup>1</sup>	_	-
SINGLE	DSC\$K_CLASS_S DSC\$K_DTYPE_F		_
DOUBLE	DSC\$K_CLASS_S DSC\$K_DTYPE_D or DSC\$K_DTYPE_G	-	_
QUADRUPLE	DSC\$K_CLASS_S DSC\$K_DTYPE_H	_	_
RECORD	_	_	_
ARRAY	DSC\$K_CLASS_A <sup>2</sup>	DSC\$K_CLASS_S DSC\$K_DTYPE_T <sup>3</sup>	_
ARRAY OF VARYING OF CHAR	DSC\$K_CLASS_VSA DSC\$K_DTYPE_VT	_	_
Conformant ARRAY	DSC\$K_CLASS_A <sup>2</sup>	DSC\$K_CLASS_S DSC\$K_DTYPE_T <sup>3</sup>	DSC\$K_CLASS_A
Conformant ARRAY OF VARYING OF CHAR <sup>4</sup>	DSC\$K_CLASS_VSA DSC\$K_DTYPE_VT	_	DSC\$K_CLASS_VSA DSC\$K_DTYPE_VT
VARYING OF CHAR	DSC\$K_CLASS_VS DSC\$K_DTYPE_VT	_	_
Conformant VARYING OF CHAR	DSC\$K_CLASS_VS DSC\$K_DTYPE_VT	_	DSC\$K_CLASS_VS DSC\$K_DTYPE_VT
SET	DSC\$K_CLASS_S DSC\$K_DTYPE_Z	_	_
FILE	DSC\$K_CLASS_S	DSC\$K_DTYPE_Z	_
Pointer	DSC\$K_CLASS_S DSC\$K_DTYPE_LU	_	_
PROCEDURE or FUNCTION	DSC\$K_CLASS_S DSC\$K_DTYPE_BPV	_	Bound procedure value by reference

<sup>&</sup>lt;sup>1</sup> Depends on size of type

<sup>&</sup>lt;sup>2</sup> Depends on component type

<sup>&</sup>lt;sup>3</sup> Only if PACKED ARRAY OF CHAR

 $<sup>^4\,\</sup>mbox{Component}$  type can be a conformant VARYING schema

Table 3-3: (Cont.) Parameter Descriptors

Parameter Type	Descriptor Class and Type		
7.	CLASS_A	CLASS_NCA	CLASS_S
Ordinal	_		DSC\$K_CLASS_S1
SINGLE	_	_	DSC\$K_CLASS_S DSC\$K_CLASS_S DSC\$K_DTYPE_F
DOUBLE	<del>-</del>	-	DSC\$CLASS_S DSC\$DTYPE_D or DSC\$DTYPE_G
QUADRUPLE	_	_	DSC\$CLAS_S DSC\$DTYPE_H
RECORD			_
ARRAY	DSC\$K_CLASS_A <sup>2</sup>	DSC\$K_CLASS_NCA <sup>2</sup>	DSC\$K_CLASS_S DSC\$K_DTYPE_T <sup>3</sup>
ARRAY OF VARYING OF CHAR	_	_	_
Conformant ARRAY	DSC\$K_CLASS_A <sup>2</sup>	DSC\$K_CLASS_NCA <sup>2</sup>	DSC\$K_CLASS_S DSC\$K_DTYPE_T <sup>3</sup>
Conformant ARRAY OF VARYING OF CHAR <sup>4</sup>	-	_	_
VARYING OF CHAR	_	_	_
Conformant VARYING OF CHAR	-	-	_
SET	_	_	DSC\$K_CLASS_S
FILE	_	_	CLASS_S DTYPE_Z
Pointer	-	_	CLASS_S DTYPE_LU
PROCEDURE of FUNCTION	_		_

<sup>&</sup>lt;sup>1</sup>Depends on size of type

<sup>&</sup>lt;sup>2</sup>Depends on component type

 $<sup>^3\</sup>mathrm{Only}$  if PACKED ARRAY OF CHAR

<sup>&</sup>lt;sup>4</sup>Component type can be a conformant VARYING schema

#### 3.2.3.1 **%STDESCR Mechanism Specifier**

When you use the %STDESCR mechanism specifier, the compiler generates a fixed-length descriptor of a character-string variable and passes its address to the called routine. Only items of the following types can have the %STDESCR specifier on the actual parameter: character-string constants, varying string expressions, packed character arrays with lower bounds of 1, and packed conformant character arrays with indexes of an integer or integer subrange type.

If the actual parameter is a variable of type PACKED ARRAY OF CHAR, %STDESCR implies variable semantics (see Section 3.2.1.2). Otherwise, %STDESCR implies foreign semantics (see Section 3.2.1.3). If the actual parameter is not modified by the called external routine, the corresponding %STDESCR formal parameter should be declared READONLY.

The following function declaration requires two fixed-length string descriptors as parameters.

```
TYPE
   Msgtype = PACKED ARRAY[1..80] OF CHAR;
   Devtype = PACKED ARRAY[1..6] OF CHAR;
VAR
   Message : Msgtype;
   Terminal : Devtype;
   Status : INTEGER:
[ASYNCHRONOUS, EXTERNAL (SYS$BRDCST)] FUNCTION $BRDCST
   (%STDESCR MSGBUF : PACKED ARRAY [$11..$u1:INTEGER] OF CHAR:
   %STDESCR DEVNAM : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED O:
   %IMMED FLAGS : INTEGER := %IMMED O;
   %IMMED CARCON : INTEGER := %IMMED 32)
   : INTEGER:
   EXTERNAL:
Status := $BRDCST (Message, Terminal);
```

The actual parameters Message and Terminal are passed by string descriptor to the formal parameters MSGBUF and DEVNAM, respectively.

#### 3.2.3.2 **%DESCR Mechanism Specifier**

When you use the %DESCR mechanism specifier, the compiler generates a descriptor for an ordinal, real, or array variable and passes its address to the called routine. The type of the %DESCR parameter can be any predefined ordinal or real type, a VARYING string, or an array (packed or unpacked, fixed or conformant) of a predefined ordinal or real type. External routines written in high-level languages or contained in the VAX Run-Time Library often require such descriptors.

If the actual parameter is a variable, %DESCR implies variable semantics (see Section 3.2.1.2). If the actual parameter is a variable or an expression enclosed in parentheses, %DESCR implies foreign semantics (see Section 3.2.1.3). If the actual parameter is not modified by the called external routine, the corresponding formal %DESCR parameter should be declared READONLY.

The following function declaration requires a varying-string descriptor as its parameter.

```
Vary = VARYING[30] OF CHAR;
  Obj_String : Vary;
  Times_Found : INTEGER;
[EXTERNAL] FUNCTION Search_String
  (%DESCR String_Val : Vary)
  : BOOLEAN:
  EXTERNAL;
IF Search_String (Obj_String)
  Times_Found := Times_Found + 1;
```

### 3.2.3.3 CLASS\_S Attribute

When you use the CLASS\_S attribute on a formal parameter, the compiler generates a fixed-length scalar descriptor of a variable and passes its address to the called routine. Only items of ordinal, real, set, pointer, and array (packed or unpacked, fixed or conformant) can have the CLASS\_S attribute on the formal parameter.

However, unlike the mechanism specifiers, the type of passing semantics used depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used, otherwise, value semantics is used.

### 3.2.3.4 CLASS\_A and CLASS\_NCA Attributes

When you use either the CLASS\_A or CLASS\_NCA attributes on a formal parameter, the compiler generates a descriptor for the array variable and passes its address to the called routine. The type of the CLASS\_A and CLASS\_NCA parameters can only be an array (packed or unpacked, fixed or conformant) of a predefined ordinal or real type.

Like the CLASS\_S attribute, the type of passing semantics used depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used, otherwise, value semantics is used.

### 3.2.4 Mechanism Specifiers on Actual Parameters

When a formal parameter is declared with a mechanism specifier, the PASCAL compiler verifies that all actual parameters passed to it are of the correct type. However, if an actual parameter has a mechanism specifier, no type checking or conversion occurs; the specifier on the actual parameter overrides that of the formal parameter. When you write a routine call, you must make sure that the actual parameters prefixed by mechanism specifiers have the correct type.

A mechanism specifier applied to an actual parameter sometimes results in ambiguities. When the actual parameter is the name of a function that has no formal parameters (or that has defaults that are being used for all its formal parameters), the ambiguity lies in whether the specifier applies to the function itself or to its result. For example:

```
[EXTERNAL] PROCEDURE Proc1
(J: REAL); EXTERNAL;

FUNCTION I
: INTEGER;
...
...
Proc1 (%REF I):
```

Because the compiler performs no type checking when function I is passed to procedure Proc1, it is unclear whether this call passes the function itself or whether it executes the function and then passes the result. (Note that the appearance of %REF before the actual parameter I inhibits the compiler from converting I to type REAL.)

The compiler resolves the ambiguity as follows:

- When the function identifier is enclosed in parentheses, the function is executed and the result is passed by the specified mechanism.
- When the function identifier is not enclosed in parentheses, the address of the function's entry mask is passed by the specified mechanism.

Thus, by using these rules, function I in the above example is passed using the by-reference mechanism. The following call, in which the function identifier is enclosed in parentheses, passes the result of I by reference:

```
Proc1 (%REF (I));
```

A similar ambiguity arises when a mechanism specifier precedes the name of a variable in an actual parameter list. %IMMED always causes a value to be passed. However, %REF, %STDESCR, and %DESCR either cause the value to be accessed directly or a local copy to be made. For example:

```
TYPE
   Arr_Type = PACKED ARRAY[1..10] OF CHAR;

VAR
   Arr : Arr_Type;

[EXTERNAL] PROCEDURE Proc2
   (Arr2 : Arr_Type); EXTERNAL;
   ...
   Proc2 (%DESCR Arr);
```

Again, the suspension of type checking makes it unclear whether this call passes the actual variable or a local copy.

The ambiguity is resolved as follows:

- When the variable identifier is enclosed in parentheses, a copy of the variable is passed by the specified mechanism.
- When the variable identifier is not enclosed in parentheses, the variable itself is accessed by the specified mechanism.

Thus, following these rules, the array Arr in the example above is passed by descriptor. The following call, in which the variable identifier is enclosed in parentheses, passes a copy of Arr by the same mechanism:

Proc2 (%DESCR (Arr));

# 3.3 Passing Parameters to PASCAL Routines

When calling a PASCAL routine from a non-PASCAL routine, you must ensure that the parameters are in the form required by the PASCAL routine. By default, VAX PASCAL requires most parameters to be passed by reference.

- When the PASCAL routine requires a value parameter, the parameter list of the calling routine must contain the address of a value. The PASCAL routine will copy the value from the passed address upon entry.
- When the PASCAL routine requires a VAR parameter, the parameter list of the calling routine must contain the address of a variable. The PASCAL routine uses the address to access the actual parameter variable. An actual parameter variable whose value can change as a result of routine execution must be passed in this manner. In addition, all files must be passed to PASCAL routines as VAR parameters.
- When the PASCAL routine requires a formal procedure or function parameter, the parameter list of the calling routine must specify the address of a bound procedure value. This process implements the VAX by-reference mechanism for a routine.
- When the PASCAL routine requires a formal conformant array or conformant VARYING parameter, the parameter list of the calling routine must contain the address of a descriptor.

# 3.4 Calling VAX/VMS System Services

You can declare any VAX/VMS system service as an external routine and then call it from a PASCAL program. When declaring a system service, you use an identifier of the following form:

#### SYS\$service-name

For example, the name of the Formatted ASCII Output (\$FAO) system service is SYS\$FAO.

Because system services are often called from condition handlers or asynchronous trap (AST) routines, you should declare system services with the ASYNCHRONOUS attribute.

When you pass parameters to a system service, you must determine, from the description of the system service, the following information about each parameter:

- 1. The mechanism used to pass the parameter
- 2. The parameter's data type
- 3. The parameter's semantics
- 4. Whether or not the parameter is optional

Parameters are passed to system services by one of three mechanisms: by reference, by immediate value, or by descriptor. Some system services require you to pass the address of a service-specific data structure (see Section 3.4.2). See the VAX/VMS System Services Reference Manual for a full description of each system service.

Because system services are not written in PASCAL, a PASCAL program usually needs special adaptations to connect with system services. VAX PASCAL provides many extensions to the syntax of routine declarations and calls that permit VAX/VMS system services and RMS services to be called easily from a PASCAL program. Two such extensions, the capabilities of supplying default values for formal parameters and of calling functions as procedures, are described in Sections 3.4.3 and 3.4.4. Other extensions that are useful with system services, such as nonpositional syntax and the use of mechanism specifiers in both formal and actual parameter lists (see also Section 3.2.4), are described in *Programming in VAX PASCAL*.

### 3.4.1 Passing Parameters to System Services

The description of each system service in the VAX/VMS System Services Reference Manual indicates the mechanism(s) and semantics by which the routine's parameters are to be passed. When passing PASCAL parameters to system services, you must use value and variable semantics or mechanism specifiers to ensure that the parameters are passed correctly. Table 3–4 lists the terms used in system service descriptions, the parameter-passing mechanism implied by these terms, and the method used to invoke each mechanism in PASCAL.

Table 3-4: Passing Parameters to System Services

System Service Description	Implied Mechanism	Method Used in PASCAL
or actual p [REFERENCE		VAR, %REF on formal or actual parameter or [REFERENCE] on formal parameter or default
Address of character-string descriptor	By-descriptor	%STDESCR or [CLASS_S] on formal parameter
Indicator, number, value, mask	By-immediate- value	%IMMED on formal or actual parameter or [IMMEDIATE] on formal parameter

Parameters passed by reference and used solely as input to a system service should be declared using PASCAL value semantics to allow actual parameters to be compile-time and run-time expressions. When a system service requires a formal parameter with a mechanism specifier, you should declare the formal parameter with the READONLY attribute to specify value semantics. Other parameters passed by reference should be declared using PASCAL variable semantics to ensure that the output data is interpreted correctly. In some cases, by-reference parameters are used for both input and output and should also be declared using variable semantics.

The following example shows the declaration of the Convert ASCII String to Binary Time (SYS\$BINTIM) system service and a corresponding function designator. The first formal parameter requires the address of a character-string descriptor with value semantics; the second requires an

address and uses variable semantics to manipulate the parameter within the service.

```
TYPE
   $QUAD = [QUAD, UNSAFE] RECORD
           LO : UNSIGNED;
           L1 : INTEGER;
           END:
VAR
   Ascii_Time : VARYING[80] OF CHAR;
   Binary_Time : $QUAD;
[ASYNCHRONOUS, EXTERNAL (SYS$BINTIM)] FUNCTION $BINTIM
   (TIMBUF : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
   VAR TIMADR : [VOLATILE] $QUAD)
   : INTEGER;
   EXTERNAL;
IF NOT ODD ($BINTIM(Ascii_Time, Binary_Time))
THEN
           (* main program *)
   BEGIN
   WRITELN ('Illegal format for time string');
   HALT;
           (* main program *)
   END;
```

### 3.4.2 Data Structure Parameters

Some system services require a parameter to be the address of a data structure that indicates a function to be performed or that holds information to be returned. Such a structure may be described as a list, a control block, or a vector. The size and POS attributes of VAX PASCAL provide an efficient method of laying out such data structures. The size attributes ensure that the fields of the data structure are of the size required by the system service. The POS attribute allows you to position the fields correctly (see *Programming in VAX PASCAL* for more information on these attributes).

For example, the Get Job/Process Information (SYS\$GETJPI) system service requires an item list consisting of an array of records of 12 bytes. By using the POS attribute on each item, you can guarantee that the fields of each record are allocated contiguously.

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM UserId (INPUT,OUTPUT);
TYPE
     Username_String = PACKED ARRAY[1..12] OF CHAR;
     Unsigned_Word = [WORD] 0..65535;
     Id_Len_Type = UNSIGNED;
     Jpi_Item
               = [BYTE(12)]
        RECORD
        Buffer_Length : [POS(0)] Unsigned_Word;
        Item_Code : [POS(16)] Unsigned_Word;
        Buffer_Address : [POS(32), UNSAFE] ^UNSIGNED;
        Return_Length_Address : [POS(64)] ^UNSIGNED;
        END;
     Jpi_Item_List = [BYTE(16)]
        RECORD
        { Item list, consisting of one item in this case. }
                      : [POS(0)] ARRAY [1..1] OF Jpi_Item;
        List
        { Longword to be set to zero to signal end of list }
        Terminator : [POS(96)] INTEGER;
        END:
VAR
     Id_Item
                : Jpi_Item_List;
     Id_Length : ^Id_Len_Type;
     Id : ^Username_String;
     Return_Code : Integer;
BEGIN
    NEW (Id):
    NEW (Id_Length);
    WITH Id_Item DO
       BEGIN
       List[1].Buffer_Length
                                   := 12:
       List[1].Item_Code
                                    := JPI$_USERNAME;
       List[1].Buffer_Address
                                   := Id;
       List[1].Return_Length_Address := Id_Length;
       Terminator := 0;
       END;
   IF GETJPI (,,,ID_ITEM) = 1
    THEN
       WRITELN ('Return_Code is: 1')
    ELSE
       WRITELN ('Return_Code is not 1');
   DISPOSE (Id);
   DISPOSE (Id_Length)
END.
```

### 3.4.3 Default Parameters

In some cases, you do not have to supply actual parameters to correspond to all the formal parameters of a system service. In VAX PASCAL, you can supply default values for such optional parameters when you declare the service. You can then omit the corresponding parameters from the routine call that invokes the service. If you choose not to supply an optional parameter, you should initialize the formal parameter with the appropriate value, using the immediate value (%IMMED) mechanism. Usually the correct default value is 0.

For example, the Cancel Timer (SYS\$CANTIM)\*system service has two optional parameters. If you do not specify values for them in the actual parameter list, you must initialize them with zeros when they are declared. The following example indicates this as it appears in STARLET.

```
[ASYNCHRONOUS, EXTERNAL(SYS$CANTIM)] FUNCTION $CANTIM (REQIDT : [IMMEDIATE] INTEGER := %IMMED O;
ACMODE : [IMMEDIATE] INTEGER := %IMMED O)
: INTEGER;
EXTERNAL;
```

A call to \$CANTIM must indicate the position of omitted parameters with a comma, unless they all occur at the end of the parameter list. For example, the following appears in STARLET:

```
$CANTIM (, PSL$C_USER);
$CANTIM (1);
$CANTIM:
```

where PSL\$C\_USER is a symbolic constant that represents the value of a user access mode, and I is an integer that identifies the timer request being canceled. Note that if you call \$CANTIM with both of its default parameters, you can omit the actual parameter list completely.

### 3.4.4 Calling System Services as Procedures

All VAX/VMS system services and VAX RMS routines are functions that return an integer condition value which indicates whether the function executed successfully. An odd-numbered condition value indicates successful completion; an even-numbered condition value indicates a warning message or failure (see the *Introduction to VAX/VMS System Routines* for further explanation of how to interpret condition values).

Your program can use the predeclared function ODD to test the function return value for success or failure. When this value is irrelevant, your program can treat the function as though it were an external procedure and ignore the return value. For example, your program can declare the Hibernate (SYS\$HIBER) system service as a function but call it as though it were a procedure:

```
[ASYNCHRONOUS, EXTERNAL (SYS$HIBER)] FUNCTION $HIBER
: INTEGER;
EXTERNAL;
...
...
$HIBER; (* Put process to sleep *)
```

Because SYS\$HIBER is expected to execute successfully, the program will ignore the integer condition value that is returned.

### 3.4.5 System Service Example

The following example illustrates several system service calls that use VAX PASCAL extensions. The program prompts for a process name and a time string. It then suspends itself and the specified process until the correct time occurs.

```
PROGRAM Suspend (INPUT, OUTPUT);
TYPE
   $QUAD = [QUAD, UNSAFE] RECORD
           LO : UNSIGNED:
           L1 : INTEGER:
           END:
   $UWORD = [WORD] 0..65535;
VAR
   Current_Time : PACKED ARRAY[1..80] OF CHAR;
   Length : $UWORD;
   Job_Name : VARYING[15] OF CHAR:
   Ascii_Time : VARYING[80] OF CHAR;
   Binary_Time : $QUAD;
[ASYNCHRONOUS, EXTERNAL (SYS$ASCTIM)] FUNCTION $ASCTIM
   (VAR TIMLEN : [VOLATILE] $UWORD := %IMMED O;
   %STDESCR TIMBUF : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED O;
   TIMADR : $QUAD := %IMMED O;
   %IMMED CVTFLG : INTEGER := %IMMED O)
   : INTEGER;
  EXTERNAL;
```

```
[ASYNCHRONOUS, EXTERNAL (SYS$BINTIM)] FUNCTION $BINTIM
   (%STDESCR TIMBUF : PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
   VAR TIMADR : [VOLATILE] $QUAD)
   : INTEGER:
   EXTERNAL;
[ASYNCHRONOUS.EXTERNAL(SYS$HIBER)] FUNCTION $HIBER
   : INTEGER;
   EXTERNAL:
[ASYNCHRONOUS, EXTERNAL (SYS$RESUME)] FUNCTION $RESUME
   (PIDADR : INTEGER := %IMMED O:
   %STDESCR PRCNAM : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED O)
   : INTEGER:
   EXTERNAL;
[ASYNCHRONOUS, EXTERNAL (SYS$SCHDWK)] FUNCTION $SCHDWK
   (PIDADR : INTEGER := %IMMED O;
   %STDESCR PRCNAM : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED O:
   DAYTIM : $QUAD:
   REPTIM : $QUAD := %IMMED O)
   : INTEGER:
  EXTERNAL:
[ASYNCHRONOUS, EXTERNAL (SYS$SUSPND)] FUNCTION $SUSPND
   (PIDADR : INTEGER := %IMMED O;
   %STDESCR PRCNAM : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED O)
   : INTEGER;
   EXTERNAL;
BEGIN
(* Print current date and time *)
$ASCTIM (TIMLEN := Length, TIMBUF := Current_Time);
WRITELN ('The current time is ', SUBSTR (Current_Time, 1,
      Length));
(* Get name of process to suspend *)
WRITE ('Enter name of process to suspend: ');
READLN (Job_Name);
(* Get time to wake process *)
WRITE ('Enter time to wake process: ');
READLN (Ascii_Time);
(* Convert time to binary *)
IF NOT ODD ($BINTIM (Ascii_Time, Binary_Time))
  BEGIN
   WRITELN ('Illegal format for time string');
   HALT;
   END;
```

```
(* Suspend process *)
IF NOT ODD ($SUSPND (PRCNAM := Job Name))
THEN
   BEGIN
   WRITELN ('Cannot suspend process');
   HALT:
   END:
(* Schedule wake up request for self *)
IF ODD ($SCHDWK (Daytim := Binary_Time))
   $HIBER (* Put self to sleep *)
ELSE
   WRITELN ('Cannot schedule wake up'):
   WRITELN ('Process will resume immediately');
   END:
(* Resume process *)
IF NOT ODD ($RESUME (PRCNAM := Job_Name))
THEN
   BEGIN
   WRITELN ('Cannot resume process');
  HALT:
  END;
END.
         (*main program *)
```

#### 3.5 **Inheriting the System Services Definitions File**

VAX PASCAL supplies a source file, SYS\$LIBRARY:STARLET.PAS, and an environment file, SYS\$LIBRARY:STARLET.PEN, which describe every VAX/VMS system service and VAX RMS routine. These files contain system service routine declarations written in VAX PASCAL. Included in these files is the following information:

- External declarations of system services and VAX RMS routines
- System symbolic names, such as system status codes and job/process information request type codes
- Data structures, such as record type definitions for record access blocks, file access blocks, extended attribute blocks, and name blocks

Use of the INHERIT attribute to obtain STARLET.PEN makes all VAX/VMS system services and VAX RMS routines available to your program so that you do not have to individually declare those you need to use. For example, the program illustrated in Section 3.4.5 would be considerably shorter if STARLET (the .PEN file type is the default) were inherited by the program. Note that the external routine declarations in

STARLET define new identifiers by which you can refer to the routines; for example, SYS\$HIBER can be referred to as \$HIBER, as in the following example:

```
[INHERIT(SYS$LIBRARY:STARLET)] PROGRAM Suspend (INPUT,OUTPUT);
TYPE
   Sys_Time = RECORD
              I,J : INTEGER;
              END;
   Unsigned_Word = [WORD] 0..65535;
VAR
   Current_Time : PACKED ARRAY[1..80] OF CHAR;
  Length : Unsigned_Word;
   Job_Name : VARYING[15] OF CHAR;
   Ascii_Time : VARYING[80] OF CHAR;
  Binary_Time : Sys_Time;
(* Print current date and time *)
$ASCTIM (TIMLEN := Length, TIMBUF := Current_Time);
WRITELN ('The current time is ', SUBSTR(Current_Time, 1, Length);
(* Get name of process to suspend *)
WRITE ('Enter name of process to suspend: ');
READLN (Job_Name);
(* Get time to wake process *)
WRITE ('Enter time to wake process: ');
READLN (Ascii_Time);
(* Convert time to binary *)
IF NOT ODD ($BINTIM (Ascii_Time, Binary_Time))
THEN
   WRITELN ('Illegal format for time string');
   HALT:
   END;
(* Suspend process *)
IF NOT ODD ($SUSPND (PRCNAM := Job_Name))
THEN
   WRITELN ('Cannot suspend process');
   HALT;
   END;
```

# 3.6 Calling VAX Run-Time Library Routines

The VAX Run-Time Library provides routines that can be called from PASCAL programs. These routines are described in the VAX/VMS Run-Time Library Routines Reference Manual.

To invoke a Run-Time Library routine from a PASCAL program, declare it as an external function and call it using a function designator. When declaring the function, you should note the following:

- The mechanism and semantics by which each parameter is passed (by immediate value, by reference, or by descriptor)
- The appropriate types for the parameters and the result

Because most routines in the VAX Run-Time Library can be called asynchronously from condition handlers and asynchronous trap (AST) routines, you should declare these routines with the ASYNCHRONOUS attribute.

The description of each routine in the VAX/VMS Run-Time Library Routines Reference Manual indicates the mechanism(s) by which the routine's parameters are to be passed. When passing PASCAL parameters to Run-Time Library routines, you use value and variable semantics or mechanism specifiers to ensure that parameters are passed correctly. Appendix C, Declaring Run-Time Library Procedures in VAX PASCAL lists the terms used in the descriptions, the mechanism implied by these terms, and the method used to invoke each mechanism in VAX PASCAL.

Most Run-Time Library routines return one of two values: (1) the result of a computation, or (2) an integer status, which indicates whether the routine completed successfully. You should always check the return status (when there is one) to make sure that the routine executed correctly. Just as with system services, an odd-numbered return status indicates success and an even-numbered return status indicates failure. You can also check for a particular return status by comparing the return status to one of the status codes defined by the system.

For example:

#### VAR

Seed\_Value : INTEGER;
Rand\_Result : REAL;

[EXTERNAL, ASYNCHRONOUS] FUNCTION MTH\$RANDOM

(VAR Seed : INTEGER)

: REAL; EXTERN;

Rand\_Result := MTH\$RANDOM (Seed\_Value);

This example uses the uniform pseudorandom number generator (MTH\$RANDOM). The seed parameter is passed by reference using variable semantics, and the result is a real number.

Some Run-Time Library routines require a variable number of parameters. For example, there is no fixed limit on the number of values that can be passed to functions that return the minimum or maximum value from a list of input parameters. The LIST attribute supplied by VAX PASCAL allows you to indicate the mechanism by which excess actual parameters are to be passed. For example:

[EXTERNAL,ASYNCHRONOUS] FUNCTION MTH\$DMIN1
 (Dlist : [LIST] DOUBLE)
 : DOUBLE;
 EXTERN;

Because the function MTH\$DIM1 has no formal parameter list, all actual parameters must be double-precision real numbers passed by reference using value semantics.

See *Programming in VAX PASCAL* for complete details on the LIST attribute.

## 3.7 Symbol Definitions Files

In addition to the STARLET.PEN environment file, VAX PASCAL provides three files containing condition symbol definitions. When you declare a system service or Run-Time Library routine, you should define the condition values by including the appropriate file in a CONST section at the beginning of your program or before the associated system service definitions. Use the %INCLUDE directive to specify the file name, as described in *Programming in VAX PASCAL*.

The three symbol definition files are described as follows:

### SYS\$LIBRARY:LIBDEF.PAS

This file contains definitions for all condition symbols from the general utility Run-Time Library routines. These symbols have the form:

LIB\$\_abc

For example:

LIB\$\_NOTFOU

### SYS\$LIBRARY:MTHDEF.PAS

This file contains definitions for all condition symbols from the mathematical routines library. These symbols have the form:

MTH\$\_abc

For example:

MTH\$\_SQUROONEG

### SYS\$LIBRARY:SIGDEF.PAS

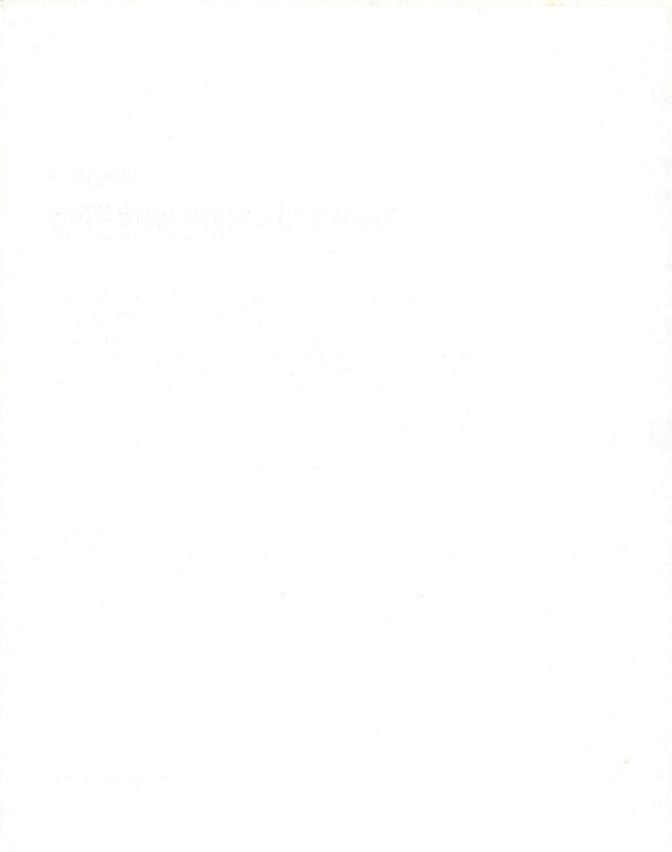
This file contains miscellaneous symbol definitions used in condition handlers. These definitions are also included in STARLET.PEN. The symbols have the form:

SS\$\_abc

For example:

SS\$\_FLTOVF

Symbol definition files can be helpful if your program does not require all the definitions contained in STARLET.PEN. For example, if your program needs definitions only for mathematical routines, you can include only the MTHDEF.PAS file.



# Input and Output with RMS

This chapter provides information about VAX PASCAL I/O in relation to VAX Record Management Services (VAX RMS or, in this chapter, simply RMS). RMS is the system VAX PASCAL uses to transfer data between an input or output device and a PASCAL program. Note that RMS uses the term "record" to denote a file component; this chapter uses the term "component" whenever possible to avoid confusion with the PASCAL record data structure. The topics covered in this chapter include:

- 1. RMS file characteristics
- 2. RMS record formats
- 3. I/O error recovery
- 4. Parameters to the OPEN and CLOSE procedures
- File sharing
- 6. RMS record locking
- 7. Use of indexed files
- 8. Programming considerations involving input and output
- 9. Interprocess communication by means of mailboxes
- 10. Remote communication by means of DECnet-VAX

### 4.1 RMS File Characteristics

A clear distinction must be made between the way files are organized and the way file components are accessed. The term "file organization" applies to the way RMS records are physically arranged on a storage device. The term "record access" refers to the method used to read components from or write components to a file, regardless of the file's organization. File organization is specified when the file is created, and cannot be changed. Record access is specified each time the file is opened, and can vary.

### 4.1.1 File Organization

VAX PASCAL supports three RMS file organizations:

- 1. Sequential
- 2. Relative
- Indexed

You specify the organization of a file with the organization parameter of the OPEN procedure, as described in Section 4.4.8.

### 4.1.1.1 Sequential Organization

A sequential file consists of components arranged in the sequence in which they are written to the file: the first component written is the first component in the file, the second component written is the second component in the file, and so on. As a result, components can be added only at the end of the file.

Sequential file organization is permitted on all devices supported by VAX RMS.

#### 4.1.1.2 **Relative Organization**

A relative file consists of numbered positions, called cells. These cells are of fixed equal length and are numbered consecutively from 1 to n, where 1 is the first cell available in the file and n is the last.

With this organization, you place a component in a file according to its cell number. The cell number is also the component's relative component number; that is, the location of the component relative to the beginning of the file. As a result, you can retrieve a component directly by specifying its relative component number.

You can add a component to, or delete one from, a file regardless of its location as long as you keep track of its relative component number.

Relative files are supported only on disk devices.

#### 4.1.1.3 **Indexed Organization**

In an indexed file, each component includes one or more key fields. Each key field establishes an ordering of the file components. Each component in an indexed file must contain at least one key. This mandatory key, called the primary key, determines the location of a component within the body of the file.

The keys of all components are collected to form one or more indexes, through which components are always accessed. The structure of the index(es) allows a program to access components in an indexed file either randomly, by specifying particular key values, or sequentially, by retrieving components with increasing key values. In addition, keyed access and sequential access can be mixed. The term Indexed Sequential Access Method (ISAM) refers to this dynamic access feature.

Indexed files are supported only on disk devices. See Section 4.8 for more information on indexed files.

### 4.1.2 Record Access

VAX PASCAL supports three RMS access methods:

- 1. Sequential
- 2. Direct
- 3. Keyed

You specify the access method of a file with the access-method parameter of the OPEN procedure, as described in Section 4.4.5.

Your choice of access method is affected by the organization of the file to be accessed. For example, sequential access can be used with sequential, relative, and indexed files; but keyed access can be used only with indexed files.

Table 4-1 shows the valid combinations of access method and file organization.

Table 4-1: Valid Combinations of Record Access Method and File Organization

File Organization	Access Method	d	
	Sequential	Direct	Keyed
Sequential	yes	yes <sup>1</sup>	no
Relative	yes	yes	no
Indexed	yes	no	yes

<sup>&</sup>lt;sup>1</sup>Fixed-length records only.

#### 4.1.2.1 **Sequential Access**

If you select sequential access for a file with sequential or relative organization, a particular component can be read only after all the components that precede it have been read; a new component can be written only at the end of the file.

With an indexed file opened for sequential access, components are read and written according to ascending primary key values. Reading from an indexed file retrieves the next component with the same or next higher specified key value. Components must be written to an indexed file with increasing primary key values.

#### 4.1.2.2 **Direct Access**

If you select direct access, you can determine the order in which components are read or written. Each series of calls to the READ procedure must be preceded by a FIND or RESET procedure, which positions the file at the specified component. Each series of calls to the WRITE procedure must be preceded by a LOCATE or REWRITE procedure.

You can use direct access on relative files and on sequential disk files that contain fixed-length RMS records. Because the direct access method uses component numbers to find file components, you can execute successive FIND or LOCATE procedures that request components in any order you choose. For example, the successive statements

```
FIND (Order_File, 24);
READ (Order_File, Order_Amt);
FIND (Order_File, 10);
READ (Order_File, Order_Amt);
```

read component 24 and then component 10.

A file for which you select direct access can be accessed sequentially as well, using GET and PUT procedures (see Programming in VAX PASCAL for more information).

#### 4.1.2.3 **Keyed Access**

If you select keyed access, you can determine the order in which components are read or updated by manipulating the keys of the components using the FINDK and RESETK procedures. The key value parameter you specify with each FINDK or RESETK procedure call is compared with index entries until the desired component is located.

When you insert a new component, the value contained in the key field of that component determines its placement in the file.

You can use keyed access only for indexed files. A file for which you select keyed access can be accessed sequentially as well, using GET and PUT procedures.

### 4.2 RMS Record Formats

RMS records are stored in one of the following formats:

- Fixed length
- Variable length
- Stream format

You specify the format of a file's components with the record-type parameter of the OPEN procedure, as described in Section 4.4.6. You can use either fixed or variable formats with any file organization; however, stream record format is supported only for sequential files.

#### Fixed-Length RMS Records 4.2.1

In a file that has fixed-length RMS records, all file components contain the same number of bytes. When you create a file that is to contain fixed-length records, the maximum size of the file components is used as the record length. This maximum size may be the default size of the file components, or be specified with the record-length parameter of the OPEN procedure (see Section 4.4.4). A file with sequential organization opened for direct access must contain fixed-length records, to allow the component number to be computed correctly.

### 4.2.2 Variable-Length RMS Records

In a file that has variable-length RMS records, the file components can contain any number of bytes, up to a specified maximum. Each component is prefixed by a count field whose value indicates the number of data bytes in that component. The count field itself requires two bytes on a disk device and four bytes on magnetic tape.

Variable-length records in relative files are actually stored in fixed-length cells. The maximum size of the file components is used as the length of the cells. This maximum size may be the default size of the components, or may be specified with the record-length parameter of the OPEN procedure (see Section 4.4.4). The value of this parameter indicates the size of the largest component that can be stored in the file.

### 4.2.3 Stream Record Format

Stream format is a record format in which records in a file are delimited by special characters or character sequences called terminators. Terminators are part of the record they delimit. The data in a stream format file is interpreted as a continuous sequence of bytes, without control information such as record counts, segment flags, or other system-supplied boundaries. Stream record format is supported for sequential files only. STREAM\_CR and STREAM\_LF indicate carriage return and line-feed, respectively. For more information on stream format, see the VAX Record Management Services Reference Manual.

# 4.3 Input/Output Error Detection

VAX PASCAL's I/O procedures accept an optional parameter called ERROR. The error-recovery parameter specifies the action the program should take if the procedure fails to execute successfully. The ERROR parameter can have one of two values, CONTINUE or MESSAGE.

When you specify ERROR := CONTINUE, the program continues to execute regardless of most error conditions encountered during execution of the procedure. If you specify ERROR := MESSAGE, an appropriate error message will be generated and execution will cease if the procedure results in an error. By default, VAX PASCAL generates an error message and ceases execution after it encounters the first error.

When you use the ERROR := CONTINUE parameter with a procedure that operates on a file, you should then use the STATUS function to determine whether execution of the procedure resulted in an error. STATUS returns an integer value to indicate the effect of the last operation on a file. A value of 0 indicates a successful operation; a value of -1 indicates that the previous operation encountered an end-of-file; a positive integer value indicates the specific error that resulted from the previous operation. (See Appendix B, Errors Returned by the STATUS and STATUSV Functions for a list of the specific error condition codes returned.)

The ERROR parameter cannot be used with the functions EOF, EOLN, or UFB, nor can it be used with references to the file buffer (f^). Therefore, you should call STATUS to check the status of the file before calling one of these functions or making a reference to the file buffer. If STATUS returns the value 0, the operation can be executed safely.

Note that the STATUS function, when used on text files, causes delayed device access to occur, resulting in the filling of the file buffer (see Section 4.9.4). Because of delayed device access, unexpected results can occur if you use the STATUS function following a READLN. Remember that a READLN procedure call actually performs a READ procedure on each variable listed as a parameter, then performs a READLN to position the file at the beginning of the next line. Therefore, a call to STATUS after a READLN actually tests whether the file was successfully positioned. To test the status of the file, STATUS fills the file buffer with the next component by performing delayed device access. If you want to test the successful reading of data from the input file, you should read the data with the READ procedure, call the STATUS function, and then perform a READLN to advance the file to the beginning of the next line.

#### **OPEN Procedure Parameters** 4.4

This section supplements the description of the OPEN procedure that appears in *Programming in VAX PASCAL*. In particular, it describes how VAX RMS affects VAX PASCAL. For more information, refer to the VAX Record Management Services Reference Manual.

The OPEN procedure can have the following two formats:

```
1.
     OPEN (file-variable
           [,file-name]
           [,file-history]
           [[,record-length]]
           [],access-method]]
           [,record-type]]
           [,carriage-control]
           [,organization]
           [,disposition]
           [[,file-sharing]]
           [].user-action[]
           [,default-file-name]
           [[,ERROR := error-recovery]])
```

```
FILE_VARIABLE := file-variable
2.

∏.FILE_NAME := file-name]

                [,FILE_HISTORY := file-history]
                [[,RECORD_LENGTH := record-length]]
                [, ACCESS_METHOD := access-method]]
                [,RECORD_TYPE := record-type]
                [,CARRIAGE_CONTROL := carriage-control] \,...)
     OPEN (
                [].ORGANIZATION := organization]
                [,DISPOSITION := disposition]
                [,SHARING := file-sharing]
                [,USER_ACTION := user-action]
                [,DEFAULT := default-file-name]
                [ ERROR := error-recovery]
```

Except for the file variable, all parameters are optional. They are VAX RMS-dependent properties, and are described in the following sections. Table 4–2 summarizes the parameters and their defaults.

If the parameter names (such as RECORD\_TYPE) are not specified, as in format 1, the parameters must be listed in the specified order. If parameter names are specified, as in format 2, the parameters can be specified in any order. You can mix the use of positional and nonpositional parameters, but once a nonpositional parameter name has been used, all the following parameter values must be nonpositional.

**Table 4–2: Summary of OPEN Procedure Parameters** 

Parameter	Parameter Values	Default
File-variable	Any file type	None, the file variable is a required parameter.
File-name	Any character string	File-variable name expression
History	OLD, NEW, READONLY, UNKNOWN	NEW (OLD, if an external file is opened using RESET)
Record-length	Any positive integer value	133 bytes for text files; for other files, parameter is ignored

Table 4-2: (Cont.) Summary of OPEN Procedure Parameters

Parameter	Parameter Values	Default
Access-method	DIRECT, KEYED, or SEQUENTIAL	SEQUENTIAL
Record-type	FIXED, VARIABLE, STREAM, STREAM_CR, STREAM_LF	VARIABLE for new text files and FILE OF VARYING; FIXED for other new files; for old files, record type established at file creation
Carriage-control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, NONE	LIST for text files and FILE OF VARYING; NOCARRIAGE for all other files. Old files use their existing carriage-control parameters
Organization	SEQUENTIAL, RELATIVE, INDEXED	SEQUENTIAL for new files; previous organization for existing files
Disposition	SAVE, DELETE, PRINT, PRINT_DELETE, SUBMIT, SUBMIT_DELETE	SAVE for named files; DELETE for files without a file-name parameter
Sharing	READONLY, READWRITE, NONE	READONLY if file history is READONLY; NONE for all other files
User-action	Function-identifier	None
Default-file-name	Any character string expression	None
Error-recovery	CONTINUE, MESSAGE	MESSAGE

### 4.4.1 File Variable

The file variable parameter is the name of the file variable associated with the file to be opened.

### 4.4.2 File Name

The file name indicates the system name of a file that is represented by a PASCAL file variable in an OPEN procedure. For the file name, you specify a character-string expression (compile- or run-time) that contains a VAX/VMS file specification or logical name. (Apostrophes are required to delimit a character-string constant or a logical name used as a file name.)

### 4.4.3 History

The history parameter indicates whether the specified file exists or must be created. A file history of NEW indicates that a new file must be created with the specified characteristics. NEW is the default value except when the file has been opened with the RESET procedure.

A file history of OLD indicates that an existing file is to be opened. An error occurs if the file cannot be found. OLD is the default value for files opened with the RESET procedure.

A file history of READONLY indicates that an existing file is being opened only for reading. An error occurs if you try to write to a file that has been opened with READONLY file history.

A file history of UNKNOWN indicates that an old file should be opened; if no old file exists, a new file is created with the specified characteristics.

### 4.4.4 Record Length

The value of the record-length parameter is a positive integer that specifies the file's maximum component size in bytes. The default value for a VAX PASCAL text file is 133 bytes. For files of other types, the maximum size of the file component is used.

By default, a file of type TEXT or VARYING OF CHAR has variable-length RMS records. The record length specified for such a file determines the length of the longest line in the file. Each line can contain any number of characters up to the record length specified. If you create a file of type TEXT or VARYING OF CHAR with fixed-length RMS records, the record length determines the exact length of each line in the file. Each line must contain the exact number of characters specified by the record length.

If you do not specify a record length for an existing file, the length specified at the file's creation is assumed.

If you create a sequential-organization file with variable-length records, the maximum record length is recorded in the file only if you specify the record-length parameter. If you wish to OPEN variable-length text files without having to know specific record lengths, you should use the following OPEN statement:

OPEN (FILE\_VARIABLE := F, HISTORY := READONLY, ERROR := CONTINUE)

When OPENing existing variable-length files, if the record-length parameter is not specified, the record-length in the file's header is used.

#### 4.4.5 **Access Method**

The access-method parameter specifies the method of access for components in the file. Access-method can have three values, SEQUENTIAL, DIRECT, or KEYED.

With the SEQUENTIAL method, you can access files that have fixedor variable-length records. The default access method is SEQUENTIAL. Using the SEQUENTIAL method allows you to use the GET, PUT, READ. RESET, REWRITE, and WRITE procedures to access files with any organization. If the file has sequential organization, you can also use the TRUNCATE procedure.

The DIRECT method allows you to use the FIND and LOCATE procedures to access files with relative organization and files with sequential organization and fixed-length records. You cannot use the DIRECT method to access a text file nor a file that has sequential organization and variable-length records.

The KEYED method allows you to access indexed files by using the FINDK and RESETK procedures to locate a specific component. You cannot open text files for KEYED access

### 4.4.6 Record Type

The record type parameter specifies the structure of the RMS records in the file. A value of FIXED indicates that all file components have the same length.

A value of VARIABLE indicates that the length of the file components can

A value of STREAM, STREAM\_CR, or STREAM\_LF indicates that the record in the file is to be delimited by special characters or character sequences called terminators. For more information on stream record format, refer to Section 4.2.3.

VARIABLE is the default record type for a new file of type TEXT or VARYING OF CHAR; other new files use FIXED as the default. For an existing file, the default is the record type associated with the file at its creation.

#### 4.4.7 **Carriage Control**

The carriage-control parameter specifies the carriage-control format for the file. A value of LIST indicates that each file component is preceded by a line feed and followed by a carriage return when the file is output to a terminal or line printer. LIST is equivalent to the VAX RMS record attribute CR. LIST is the default option for all text files and files of type VARYING OF CHAR.

The CARRIAGE or FORTRAN option indicates that the first byte of each component contains a carriage-control character. CARRIAGE or FORTRAN is equivalent to the VAX RMS record attribute FTN.

Table 4-3: **Carriage-Control Characters** 

Character	Meaning		
<b>'+'</b>	Overprinting: starts output at the beginning of the current line		
, ,	Single spacing: starts output at the beginning of the next line		
<b>'0'</b>	Double spacing: skips a line before starting output		
<b>'1'</b>	Paging: starts output at the top of a new page		
<b>'\$'</b>	Prompting: starts output at the beginning of the next line and suppresses carriage return at the end of the line		
"(0)	Prompting with overprinting: suppresses line feed at the beginning of the line and carriage return at the end of the line; note that this character is the ASCII NUL character		

The NOCARRIAGE or NONE option indicates that the component contains no carriage-control information. NONE is the default option, except for text files and files of type VARYING OF CHAR.

#### 4.4.8 **Organization**

The organization parameter specifies file organization. It can specify the values SEQUENTIAL, RELATIVE, or INDEXED. The default is SEQUENTIAL. However, if you omit the organization parameter when opening an existing file, the organization specified in the existing file is used. You may not specify an organization that differs from that of an existing file.

You may not specify RELATIVE or INDEXED organization for a text file.

# 4.4.9 Disposition

The disposition parameter specifies what is to be done with the file when it is closed. If you specify SAVE, the file is retained. SAVE is the default value for files named by the FILE\_NAME parameter.

If you specify DELETE, the file is deleted. If you specify PRINT, the file is submitted to the system line printer and is not deleted. The file is deleted after being printed only if you specify PRINT\_DELETE.

If you specify SUBMIT, the file is submitted to the batch job queue and is not deleted. The file is deleted after being processed only if you specify SUBMIT\_DELETE.

You cannot save an unnamed file because it is automatically deleted when it is closed. DELETE is the only value allowed for unnamed files.

# **4.4.10** Sharing

The sharing parameter indicates whether other programs can access the file while it is open. A value of READONLY indicates that other programs can read the file while it is open, but cannot write to it. READONLY is the default value for files that have a history of READONLY.

A value of READWRITE indicates that other programs can read and write to the file while it is open.

A value of NONE denies other programs any access to the file while it is open. NONE is the default value for files with histories NEW, OLD and UNKNOWN.

If you specify SHARING := READWRITE for an existing file with sequential organization, you must explicitly specify ORGANIZATION := SEQUENTIAL in the same OPEN procedure. Note that the components of a sequential file with READWRITE sharing are not locked (see Section 4.7).

#### 4.4.11 **User Action**

The user-action parameter allows you to access RMS facilities not explicitly available in the VAX PASCAL language by writing a function that controls the opening of the file. Inclusion of the user-action parameter causes the Run-Time Library to call your function to open the file instead of calling RMS to open it according to its normal defaults.

When an OPEN procedure is executed, the Run-Time Library uses the procedure's parameters to establish the VAX RMS File Access Block (FAB) and the Record Access Block (RAB), as well as to establish its own internal data structures. These blocks are used to transmit requests for file and record operations to RMS; they are also used to return the data contents of files, information about file characteristics, and status codes.

In order, the three parameters passed to a user-action function by the Run-Time Library are:

- 1. FAB address
- 2. RAB address
- File variable

A user-action function is usually written in VAX PASCAL and includes:

- 1. Modifications to the FAB and/or RAB (optional)
- 2. \$OPEN and \$CONNECT for existing files or \$CREATE and \$CONNECT for new files (required)
- 3. Status check of the values returned by \$OPEN or \$CREATE and \$CONNECT (required)
- 4. Storage of FAB and/or RAB values in program variables (optional)
- 5. Return of success or failure status value for the user-action function (required)

The following example shows a VAX PASCAL program that copies one file into another. The program features two user-action functions, which allow the output file to be created with the same size as the input file and to be given contiguous allocation on the storage media.

```
[INHERIT ('SYS$LIBRARY:STARLET')]
PROGRAM Contiguous_Copy (F_In, F_Out);
(* The input file F_In is copied to the output file F_Out.
  F_Out has the same size as F_In and has contiguous
  allocation. *)
TYPE
  FType = FILE OF VARYING[133] OF CHAR;
VAR
  F_In, F_Out : FType;
  Alloc_Quantity : UNSIGNED;
FUNCTION User_Open
   (VAR FAB : FAB$TYPE;
    VAR RAB : RAB$TYPE;
    VAR F : FType)
                       : INTEGER;
   VAR
     Status : INTEGER:
                  (* Function User_Open *)
   (* Open file and remember allocation quantity *)
   Status := $OPEN (FAB);
   IF ODD (Status)
   THEN
      Status := $CONNECT (RAB);
   Alloc_Quantity := FAB.FAB$L_ALQ;
   User_Open := Status;
   END;
                  (* Function User_Open *)
FUNCTION User_Create
   (VAR FAB : FAB$TYPE;
    VAR RAB : RAB$TYPE;
    VAR F : FType)
                       : INTEGER;
     Status : INTEGER;
   BEGIN
                  (* Function User_Create *)
   (* Set up contiguous allocation *)
   FAB.FAB$L_ALQ := Alloc_Quantity;
   FAB.FAB$V_CBT := FALSE;
   FAB.FAB$V_CTG := TRUE;
   Status := $CREATE (FAB);
   IF ODD (Status)
   THEN
        Status := $CONNECT (RAB);
   User_Create := Status;
                 (* Function User_Create *)
```

```
BEGIN
                  (* main program *)
(* Open files *)
OPEN (F_In,
      HISTORY := READONLY.
      USER_ACTION := User_Open);
RESET (F_In);
OPEN (F_Out.
     HISTORY := NEW,
     USER_ACTION := User_Create);
REWRITE (F_Out);
(*Copy F_In to F_Out*)
WHILE NOT EOF (F_In) DO
  BEGIN
  WRITE (F_Out, F_In_^);
  GET (F_In);
  END;
(* Close files *)
CLOSE (F_In):
CLOSE (F_Out);
END.
                  (* main program *)
```

In this example, the record types FAB\$TYPE and RAB\$TYPE are defined in SYS\$LIBRARY:STARLET, which the program inherits. The function User\_Open is called as a result of the OPEN procedure for the input file F\_In. The function begins by opening the file with the RMS service \$OPEN. If \$OPEN succeeds, the value of Status is odd; in that case, \$CONNECT is performed. The allocation quantity contained in the FAB.FAB\$L\_ALQ field of the FAB is assigned to a variable so that this value can be used in the second user-action function. User\_Open is then assigned the value of Status (in this case, TRUE), which is returned to the main program.

The function User\_Create is called as a result of the OPEN procedure for the output file F\_Out. The function assigns the allocation quantity of the input file to the FAB.FAB\$L\_ALQ field of the FAB, which contains the allocation size for the output file. The FAB field FAB.FAB\$V\_CBT is set to FALSE to disable the request that file storage be allocated contiguously on a "best try" basis. Then, the FAB field FAB.FAB\$V\_CTG is set to TRUE so that contiguous storage allocation is mandatory. Finally, the RMS service \$CREATE is performed. If \$CREATE is successful, \$CONNECT will be done and the function return value will be that of \$CREATE.

Once the OPEN procedures have been performed successfully, the program can then accomplish its main task, that of copying the input file F\_In to the output file F\_Out, which is the same size as F\_In and has contiguous allocation. The last step in the program is to close the files.

## 4.4.12 Default File Name

The DEFAULT file name allows you to specify default values for portions of a file name. You can specify a character-string expression (either compile- or run-time) that contains the portions of the VAX/VMS file specification you want to have defaulted. (Apostrophes are required to delimit a character-string constant used as a default file name.)

If you supply a full file name specification with the FILE\_NAME parameter, then the DEFAULT parameter will be ignored. If, however, you supply only a partial file name specification with the FILE\_NAME parameter, the rest of the file name specification will be constructed from the DEFAULT parameter, if one has been specified. If no DEFAULT parameter was specified, then RMS uses the FILE\_NAME, and RMS defaults to construct the file name specification. An example using the DEFAULT keyword follows:

## Example

The preceding OPEN statement will OPEN the file called '[ANOTHER.DIR]FOO.BAR'. For more information on default file names, see the VAX Record Management Services Reference Manual.

# 4.4.13 Error Recovery

The error-recovery parameter specifies the action the program should take if the OPEN procedure fails to execute successfully. If you specify ERROR := CONTINUE, the program continues to execute regardless of any error conditions encountered while the specified file was being opened.

If you specify ERROR := MESSAGE, the appropriate error message will be generated and execution will cease if the procedure results in an error. By default, VAX PASCAL generates an error message and ceases execution after it encounters the first error. See Section 4.3 for more information on error detection.

# 4.5 **CLOSE Procedure Parameters**

This section supplements the description of the CLOSE procedure that appears in *Programming in VAX PASCAL*.

The CLOSE procedure has the following format:

Except for the file variable, all parameters are optional. The optional parameters are VAX RMS-dependent properties, and are described in the following section.

# 4.5.1 Disposition

The disposition parameter specifies what is to be done with the file when it is closed. The parameter values and the defaults are the same as those for the disposition parameter in the OPEN procedure (see Section 4.4.9).

If a disposition value is specified in the CLOSE procedure, it overrides the value of the disposition parameter in the OPEN procedure.

# 4.5.2 User Action

The user-action parameter of the CLOSE procedure is similar to that of the OPEN procedure in that it allows you to access RMS facilities not directly available in VAX PASCAL by writing a function that controls the closing of the file. Including the user-action parameter causes the Run-Time Library to call your function to close the file instead of calling RMS to close it according to its normal defaults.

When a CLOSE procedure is executed, the Run-Time Library uses the procedure's parameters to modify the VAX RMS File Access Block (FAB) and the Record Access Block (RAB), as well as using its own internal data structures, if necessary.

In order, the three parameters passed to a user-action function by the Run-Time Library are:

- FAB address
- 2. RAB address
- 3. File variable

A user-action function is usually written in VAX PASCAL and includes:

- 1. Modifications to the FAB and/or RAB (optional)
- 2. \$CLOSE when PASCAL I/O was performed (required)
- 3. Status check of the value returned by \$CLOSE (required)
- 4. Return of success or failure status value for the user-action function (required)

By including a user-action function in the CLOSE procedure, you can change file disposition, change the file name, rewind tape volumes, and deallocate unused space at the end of a file.

#### 4.5.3 **Error Recovery**

The error-recovery parameter specifies the action the program should take if the CLOSE procedure fails to execute successfully. The parameter values and the default action are the same as those for the error-recovery parameter in the OPEN procedure (see Section 4.4.13).

# 4.6 File Sharing

Through the RMS file-sharing capability, a file can be accessed by more than one open program at a time or by the same program through more than one file variable. There are two kinds of file sharing—read sharing and write sharing. Read sharing occurs when several programs are reading a file at the same time. Write sharing takes place when at least one program is writing a file and at least one other program is either reading or writing the same file.

The extent to which file sharing can take place is determined by three factors: the type of device on which the file resides, the organization of the file, and the explicit information supplied by the user. These elements affect file sharing in the following ways:

## Device type

Sharing is possible only on disk files, since other files must be accessed sequentially.

## 2. File organization

All three file organizations permit read sharing on disk files. In addition, relative and indexed files allow write sharing. Sequential files allow a restricted form of write sharing: only one program at a time can be writing to a sequential file. However, because the file system does not protect against simultaneous updating of sequential file records, your program must ensure that access conflicts cannot occur.

# 3. Explicit user-supplied information

Whether or not file sharing actually takes place depends on information provided by the user for each program that accesses the file. In VAX PASCAL programs, this information is supplied by the values of the SHARING parameter in the OPEN procedure, READONLY and READWRITE (see Section 4.4.10).

Read sharing can occur when SHARING := READONLY is specified by all programs that access the file. Write sharing is accomplished when all programs specify SHARING := READWRITE.

Programs that specify SHARING := READONLY or SHARING := READWRITE can access a file simultaneously; however, file sharing can fail under certain circumstances. For example, a program without either of these parameters will fail when it attempts to open a file currently being accessed by some other program. Or, a program that specifies SHARING := READONLY or SHARING := READWRITE can fail to open

a file because a second program with a different specification is currently accessing that file.

When two or more programs are write-sharing a file, each program should use one of the error-processing mechanisms described in Chapter 5, Error Processing and Condition Handlers. Use of such controls prevents program failure due to a record-locking error. Section 4.7 discusses error detection and the handling of locked records.

# 4.7 Record Locking

The RMS record-locking facility, along with the logic of the program, ensures that a program can add, delete, or update a component without having to do a synchronization check to determine whether that component is currently being accessed by another process. In other words, record locking prevents two processes from accessing the same component simultaneously.

When a program opens a relative or indexed file and specifies SHARING := READWRITE, RMS locks each component as it is accessed. When a component is locked, any program that attempts to access it fails and a record-locked error results. A subsequent I/O operation on the file variable unlocks the previously accessed component. Thus, at most one component is locked for each file variable.

A VAX PASCAL program can explicitly unlock a component by executing the UNLOCK procedure. To minimize the time during which a component is locked against access by other programs, the UNLOCK procedure should be used in programs that retrieve components from a shared file but that do not attempt to update them. VAX PASCAL requires that a component be locked before a DELETE or an UPDATE procedure can be executed.

Record locking is not provided for sequential files. They can be writeshared, however, as long as you provide the logic necessary to synchronize the simultaneous reading and writing.

### 4.8 **Using Indexed Files**

You can access indexed files with both sequential and keyed access methods. Sequential reading retrieves consecutive components, which are sorted according to the specified key field. Keyed access, on the other hand, permits random component selection according to the value of a particular key field. Once you select a component by key, a sequential read retrieves components with ascending key values, beginning with the key field value of the initial FINDK or RESETK procedure. When you specify the KEYED access method in the OPEN procedure (see Section 4.4.5), you enable both sequential and keyed access to the indexed file.

Indexed organization is especially suitable for manipulating complex files in which you want to select components based on one of several criteria. For example, a mail-order firm could use an indexed file to store its customer list. Key fields might designate a unique customer order number, the customer's zip code, and the item ordered. Reading sequentially based on the zip code key would produce a mailing list already sorted by zip code, while reading sequentially based on the item-ordered key would give a list of customers sorted according to the product ordered.

#### 4.8.1 Creating an Indexed File

Within a VAX PASCAL program, you can create an indexed file in two ways:

- 1. With the VAX PASCAL OPEN procedure
- 2. With the RMS file definition utility (\$CREATE/FDL)

You can use the OPEN procedure to specify common file characteristics; you must use the \$CREATE/FDL utility to select features not directly supported by PASCAL. However, any indexed file created with \$CREATE /FDL can still be accessed by PASCAL I/O statements. An indexed file can be created with the OPEN procedure only when the file components are of type RECORD. To create an indexed file with components of any other VAX PASCAL type, use \$CREATE/FDL or a user-action function in your program.

To define the key fields of an indexed file, you use the KEY attribute (see Programming in VAX PASCAL). The KEY attribute can be applied only to fields of PASCAL record types. If the record type has variants, the variant parts should not contain key fields.

One of the key fields, called the primary key, is identified as key number 0 and must be present in every file component. Additional keys, called alternate keys, can also be defined; they are numbered from 1 through a maximum of 254. While an indexed file may have as many as 255 key fields defined, in practice few applications require more than three or four key fields.

When you design an indexed file, you decide which byte positions within each component are the key fields. The key data types supported by VAX PASCAL are PACKED ARRAY OF CHAR and the ordinal types. If a key field of an ordinal type has a size attribute, it must be allocated one of the following: a signed word, a signed longword, an unsigned byte, an unsigned word, or an unsigned longword. Without a size attribute, the VAX PASCAL compiler will correctly allocate key fields by default.

The following example of a mail-order firm shows how you might define key fields in a file component:

```
TYPE

Mail_Order : RECORD

Order_Num : [KEY(0)] INTEGER;

Name : PACKED ARRAY[1..20] OF CHAR;

Address : PACKED ARRAY[1..20] OF CHAR;

City : PACKED ARRAY[1..19] OF CHAR;

State : PACKED ARRAY[1..2] OF CHAR;

Zip_Code : [KEY(1)] PACKED ARRAY[1..5] OF CHAR;

Item_Num : [KEY(2)] INTEGER;

Shipping : REAL;

END;
```

Given this record type definition, you could use the following OPEN statement to create an indexed file:

```
OPEN (Order_File,
    'CUSTOMERS.DAT',
    HISTORY := NEW,
    ACCESS_METHOD := KEYED,
    ORGANIZATION := INDEXED);
```

Order\_File : FILE OF Mail\_Order; Order\_Rec : Mail\_Order;

The type definition establishes three key fields, one primary key and two alternate keys, for the record type Mail\_Order. The file Order\_File that is opened by the OPEN procedure is declared as a file of Mail\_Order records. The OPEN parameters define the file as an indexed file opened for keyed access.

The following RMS default key attributes apply to the creation of an indexed file:

- 1. Primary key values cannot be changed when a record is rewritten.
- 2. Primary key values cannot be duplicated; that is, no two records can have the same primary key value.
- 3. Alternate keys may both be changed and have duplicates.

You can use the \$CREATE/FDL utility or a user-action function to override these defaults and to specify other values not supported by VAX PASCAL, such as null key values, key names, and key data types other than those discussed here.

The VAX Record Management Services Reference Manual has more information on indexed file options. Use of the \$CREATE/FDL utility is explained in detail in the VAX Record Management Services Reference Manual.

#### 4.8.2 **Current Component and Next Component Pointers**

RMS maintains two pointers into an open indexed file: the "next component" pointer and the "current component" pointer. When you reset an indexed file, the current component pointer indicates the file component with the lowest primary key value. Subsequent GET operations cause the current component pointer to indicate the component with the next higher key value in the same key field. If duplicate key values occur, the components are retrieved in the order in which they were written.

The current component is always the one most recently locked by a successful call to GET, RESET, RESETK, FIND, or FINDK; it is undefined until the file is either locked by one of these procedures or operated upon by any other file operation. The UPDATE and DELETE procedures (see Sections 4.8.5 and 4.8.6) operate on the current file component; thus, if one of these procedures is called while the current component is undefined or unlocked, an error results.

The next component pointer indicates the component to be retrieved by the next GET operation.

#### 4.8.3 **Writing Indexed Files**

You can write components to an indexed file with either the WRITE or the PUT procedure. Each WRITE or PUT inserts a new component in the file and updates the index(es) so that the new component appears in the correct order for each key field.

In the mail-order file example of Section 4.8.1, you could add a new component to the file with either of the following sets of statements:

```
WRITE (Order_File, Order_Rec);
Order_File := Order_Rec;
PUT (Order_File);
```

It is possible to write two or more components with the same key value. Whether or not this duplicate key situation is allowed depends on the file characteristics specified when the file was created. By default, VAX PASCAL allows indexed files to have duplicate alternate keys but prohibits duplicate primary keys (see Section 4.8.1). If duplicate keys are present in a file, the components with equal keys are retrieved in the order in which they were originally written.

For example, assume that five components are written to an indexed file in the following order (for clarity, only key fields are shown):

order_num	zip_code	item_num	
1023	70856	375	
942	02163	2736	
903	14853	375	
1348	44901	1047	
1263	33032	690	

If the file is later opened and read sequentially by primary key (Order\_ Num), then the sorted order of the components is unaffected by the duplicated Item\_Num key:

order_num	zip_code	item_num	
903	14853	375	
942	02163	2736	
1023	70856	375	
1263	33032	690	
1348	44901	1047	

If the file is read along the second alternate key (Item\_Num), however, the sort order is affected by the duplicate key:

order_num	zip_code	item_num	
1023	70856	375	
903	14853	375	
1263	33032	690	
1348	44901	1047	
942	02163	2736	

Notice that the components containing the same key value (375) were retrieved in the order in which they were written to the file.

#### 4.8.4 **Reading Indexed Files**

You can read components of an indexed file with either RESETK or FINDK procedures. These procedures position the file pointers (see Section 4.8.2) at a particular component, determined by the key number, the key value, and the match type. Once you retrieve a component by key, you can use READ or GET statements to retrieve components with increasing key

The following example prints the order number and zip code of each file component that has a zip code greater than or equal to 10000 but less than 50000:

```
FINDK (Order_File, 1, '10000', GEQ);
Continue := TRUE;
WHILE Continue AND NOT UFB (Order_File) DO
  READ (Order_File, Order_Rec);
   IF Order_Rec.Zip_Code < '50000'
     WRITE (Report_File, 'Order number', Order_Rec.Order_Num,
           'has zip code', Order_Rec.Zip_Code)
      Continue := FALSE;
   END:
```

If you want to detect whether there is data available to be read from an indexed file, you can test the status of the file with the UFB function as in the above example. If UFB is TRUE, no record can be read; if UFB is FALSE, the READ procedure will successfully retrieve a record (see Programming in VAX PASCAL for a full description of the UFB function).

#### 4.8.5 **Updating Components**

To update existing components of an indexed file, use the UPDATE procedure. You cannot replace an existing component simply by writing it again; a WRITE or PUT procedure attempts to add a new component.

An update operation is accomplished in two steps. First, you must lock the component, thereby making it the current component. Next, you must execute the UPDATE procedure. For example, to update the component containing Order\_Num 903 (see prior examples) so that the Name field becomes 'Theodore Zinck', you might use the following statements:

```
FINDK (Order_File, 0, 903, EQL);
Order_Rec.Name := 'Theodore Zinck';
UPDATE (Order_File);
```

When you update a component, RMS defaults allow you to change the values of any key fields except the primary key. You can change RMS defaults by using \$CREATE/FDL or a user-action function to allow primary key values to change or to prohibit alternate key values from changing.

The following example shows the updating of a file to add shipping charges to all orders mailed to areas with zip codes less than 50000:

```
RESETK (Order_File, 1);
Continue := TRUE;
WHILE Continue AND NOT UFB (Order_File) DO
    IF Order_File^.Zip_Code < '50000'
    THEN
        BEGIN
        Order_File^.Shipping := 1.00;
        UPDATE (Order_File);
        GET (Order_File);
        END
    ELSE
        Continue := FALSE;</pre>
```

This example uses the UPDATE procedure to change the value in the field Order\_File^. Shipping. Because components must be locked before the update can occur, the GET procedure is used here to retrieve the next component. GET locks the retrieved component; the READ procedure does not.

## 4.8.6 Deleting Components

To delete components from an indexed file, use the DELETE procedure. The DELETE and UPDATE procedures are similar in that, before a component can be operated upon, it must first be locked.

The following example deletes the second component in the file with Item\_Num 375 (see prior examples):

Deletion removes a component from all defined indexes in the file.

#### 4.8.7 **Exception Conditions**

When using indexed files, you can expect to encounter certain exception conditions, which usually result from such operations as:

- Attempting to read a file component that does not exist
- Attempting to write a record that invalidly duplicates a key
- Attempting to read a locked file component

If the component specified by a FINDK procedure does not exist, an error does not occur; UFB simply becomes TRUE. Your program should test the status of UFB for this possibility.

Other exception conditions can be handled by the STATUS function and by the error-recovery parameter that is available with I/O procedures. The STATUS function indicates the status of a file following the last operation performed on it. The error-recovery parameter indicates the action to be taken should an I/O procedure fail. You can also write condition handlers (see Chapter 5, Error Processing and Condition Handlers) to allow your program to recover from I/O errors, although that method is not the simplest.

The following example shows how the use of the STATUS function and the error-recovery parameter allow a program to recover from an exception condition:

```
READ (Flight, No_of_Passengers);
1: FINDK (Reservation_File, O, Flight, EQL, ERROR := CONTINUE);
IF STATUS (Reservation_File) = PAS$K_FAIGETLOC
THEN
  BEGIN
      WRITE ('Reservation record locked. Try again',
         '(Yes or No) : ');
      READ (Response);
      If Response = Yes
      THEN
         GOTO 1:
  END
ELSE
   IF STATUS (Reservation_File) = 0
   THEN
      BEGIN
         Reservation_File^.Passenger_Count :=
            Reservation_File^.Passenger_Count + No_of_Passengers;
         UPDATE (Reservation_File);
      END
      WRITE ('Error accessing Reservation_File');
```

In this example, PAS\$K\_FAIGETLOC is the symbolic name that represents the record-locked error code. Following the FINDK procedure, STATUS tests whether the specified component of Reservation\_File was locked. Because the ERROR parameter specifies that execution should continue in the event of an error, the FINDK procedure is repeatedly attempted until the record is no longer locked. When STATUS returns a value of 0, indicating that FINDK was successful, the passenger count field is increased. When STATUS returns a value other than 0 or the value of PAS\$K\_FAIGETLOC, the program prints an error message.

## **Input/Output Considerations on Text Files** 4.9

The RMS unit of transfer is a file structure called a record, which should not be confused with the PASCAL type RECORD. For all PASCAL file types other than text files, there is a one-to-one correspondence between a PASCAL file component and an RMS record. For example, each RMS record that constitutes a FILE OF INTEGER contains the representation of one integer value. A component of a text file is a single character; however, an RMS record in a text file is equivalent to a complete line of characters as terminated by an end-of-line marker.

The processing of input and output in VAX PASCAL requires special considerations, as described in the following sections.

#### 4.9.1 **Text File Buffering**

RMS always deals with complete records. When characters are being transferred between a physical I/O device and a PASCAL text file, the record is not complete until an end-of-line marker is detected. Therefore, as the characters are read or written, they are stored in an internal line buffer until a complete record can be transferred through RMS. The line buffer contains characters of a partially processed line of text.

Because RMS records correspond exactly to the components of non-text files, no buffering is necessary for files of any type other than TEXT.

## 4.9.2 FILE OF CHAR

The file type FILE OF CHAR differs from the predefined file type TEXT in that FILE OF CHAR allows a single character to be the unit of transfer between a program and its associated I/O devices. Single-character RMS records are always read with the READ procedure, and must be read exclusively into variables of type CHAR, including CHAR components of structured variables. The EOLN, READLN, and WRITELN routines cannot be used on files of type FILE OF CHAR because such files, unlike text files, do not have end-of-line markers.

# 4.9.3 Prompting of Text Files

A prompt is a line of text that appears on the terminal screen when a partial RMS record is written to a terminal output file. Normally, when a WRITE procedure is performed on a text file connected to a terminal, the individual characters are accumulated in the line buffer until a subsequent WRITELN procedure is executed. In effect, WRITELN generates an endof-line marker and thereby completes an RMS record. When the record is completed or the file is closed, a full line of characters is written to the specified text file.

The VAX PASCAL run-time system is capable of dealing with partial records, however, when characters are being written to a terminal output file opened with the LIST carriage-control option (LIST is the default; see Section 4.4.7), partial records are written to the terminal before input is transferred from any terminal to the line buffer of a text file. In this situation, PASCAL searches for all text files opened for output on terminals; it then writes to those files any partial records contained in the files' respective line buffers. These partial records, called prompts, appear on the terminal screens. You respond to a prompt by typing a line of input data terminated by a break character. Usually a line of input from the terminal ends with a RETURN, but you can type any VMS break character (including ESCAPE) to mark the end of a line.

You will also be prompted for input when your program tests for either end-of-line (with the EOLN function) or end-of-file (with the EOF function), and the internal line buffer of the tested file is empty. To perform the test, partial lines of output are transferred from each line buffer to the corresponding terminals. The file's line buffer is filled by the input you type in response to the prompt. If you immediately type CTRL/Z, an end-of-file condition exists on the terminal and EOF will return TRUE. If you type any other VMS break character, the internal line buffer will

contain an end-of-line marker and EOLN will return TRUE. Any other input you type will be placed in the internal line buffer.

Prompting does not occur on terminal output files that have a carriagecontrol option other than LIST or that were opened by a user-action function.

#### 4.9.4 **Delayed Device Access to Text Files**

The PASCAL standard specifies that the file buffer must always contain the next file component that the program will process. This requirement can cause problems when the input to the program depends on the output most recently generated. To alleviate these problems in the processing of text files, VAX PASCAL uses a technique called delayed device access, also known as "lazy lookahead."

As a result of delayed device access, input is not retrieved from a physical file device and inserted in the file buffer until the program is ready to process it. The file buffer is filled when the program makes the next reference to the file. A reference to the file consists of any use of the file buffer variable, including its implicit use in the GET, READ, and READLN procedures, or any explicit test for the status of the file, namely, the EOF, EOLN, STATUS, and UFB functions.

The RESET procedure, which is required when any text file is opened for input, initiates the process of delayed device access. (Note that RESET is done automatically on the predeclared file INPUT.) RESET expects to fill the file buffer with the first component of the file. However, because of delayed device access, data is not supplied from the input device to fill the file buffer until the next reference to the file occurs.

When writing a program in which a text file will supply the input, you should be aware that delayed device access will occur. Since RESET initiates delayed device access and since EOF and EOLN cause the file buffer to be filled, you should place the first prompt for input before any tests for EOF or EOLN. The data you enter in response to the prompt is retained by the file device until you make another reference to the input file.

The following example illustrates the use of prompts in the reading of input data and delayed device access:

```
VAR
Purch_Amount : REAL;

...

WRITE ('Enter amount of purchase or <CTRL/Z>: ');
WHILE NOT EOF DO
BEGIN
READLN (Purch_Amount);
WRITE ('Enter amount of purchase or <CTRL/Z>: ');
END:
```

The first reference to the file INPUT is the EOF test in the WHILE statement. When the test is performed, the PASCAL run-time system attempts to read a line of input from the text file. Therefore, in this program, it is very important to prompt for the amount of purchase before testing for EOF. If you respond to the prompt by typing CTRL/Z, EOF returns TRUE. If you respond by entering a purchase amount, EOF returns FALSE.

Suppose you respond to the first prompt for input by typing a real number. Access to the input device is delayed until the EOF function makes the first reference to the file INPUT. The EOF function causes a line of text to be read into the internal line buffer. The subsequent READLN procedure reads the input value from the line of text and assigns it to the variable Purch\_Amount. The final statement in the WHILE loop is the request for another input value. The WHILE loop is executed until EOF detects the end-of-file marker.

A sample run of a program containing this loop might be:

```
$ RUN PURCH
Enter amount of purchase or <CTRL/Z>: 7.95
Enter amount of purchase or <CTRL/Z>: 6.49
Enter amount of purchase or <CTRL/Z>: 19.99
Enter amount of purchase or <CTRL/Z>: ^2
```

The following program fragment illustrates a method of writing the same loop that does not take into account delayed device access and therefore produces incorrect results:

```
WHILE NOT EOF DO
BEGIN
WRITE ('Enter amount of purchase or <CTRL/Z>: ');
READLN (Purch_Amount);
FUN.
```

The EOF test at the beginning of the loop causes the file buffer to be filled. However, because no input has yet been supplied, the prompt will not appear on the terminal screen until you have supplied input to fill the INPUT file buffer.

A sample run of a program containing this loop might be:

```
$ RUN PURCHASE
7.95
Enter amount of purchase or <CTRL/Z>: 6.49
Enter amount of purchase or <CTRL/Z>: 19.95
Enter amount of purchase or <CTRL/Z>: ^Z
```

Note that the prompt always appears after you have typed a value for Purch\_Amount.

## 4.10 **Interprocess Communication: Mailboxes**

The exchange of data between processes is often useful; for example, such an exchange can synchronize execution or send messages. A mailbox is a record-oriented, pseudo-I/O device that allows data to be passed from one process to another. Mailboxes are created by the Create Mailbox (SYS\$CREMBX) system service. See the VAX/VMS System Services Reference Manual for information about using SYS\$CREMBX.

Data transmission by means of mailboxes is synchronous; that is, a program that uses a mailbox to read information from another process must wait until that process has sent the information to the mailbox. Likewise, a program that writes information into a mailbox must wait until the other process has read the information before execution can continue. When two processes exchange information between text files, delayed device access occurs as it always does for text files. When information is exchanged between non-text files, file components are not buffered; as soon as one process writes a component, the other process can read it.

When a process closes a mailbox file that it opened for writing (that is, the mailbox was opened without HISTORY:=READONLY), an end-of-file "message" is written to the mailbox. The next process to read from the mailbox will receive an end-of-file condition. If you plan only to read from a mailbox, you should specify HISTORY := READONLY when you open it to prevent unwanted end-of-file conditions when the mailbox is closed.

# 4.11 Communicating with Remote Computers: Networks

If your computer is a node in a DECnet network, you can use VAX PASCAL I/O procedures to communicate with other nodes in the same network. These procedures allow you to exchange data with a program at the remote computer (task-to-task communication) and to access files at the remote computer (resource sharing).

Both task-to-task communication and resource sharing between systems are transparent; that is, these intersystem exchanges do not appear to be different from local interprocess and file-access exchanges.

To communicate across the network, you must specify a node name as the first element of a file specification. For example:

```
BOSTON::DISK$USER:[SMITH]TEST.DAT:21
```

Remote task-to-task communication requires a special form of file specification. You must use the notation TASK= in place of the device name and supply the task name, enclosing this information in quotation marks (""). For example:

```
BOSTON:: "TASK=UNA"
```

The example specifies a task using the command file UNA.COM on the BOSTON node of the network.

The following program fragment shows how messages can be received from a remote program by means of VAX PASCAL I/O procedures:

```
OPEN (FILE_VARIABLE := Netjob,
      FILE_NAME := 'BOSTON::"TASK=UNA"',
      HISTORY := OLD);
RESET (Netjob);
Rdat := Netjob^;
Net_Proc (Rdat, Wrtdat);
CLOSE (Netjob);
```

The effect of these statements is first to establish a link with a job executing the command file UNA.COM at the node BOSTON and then to receive a component from the file variable associated with the remote program. The variable Rdat contains the data. Then, the procedure Net\_ Proc is called to process the data, and the link is broken.

The next example shows how you can write a remote file using VAX PASCAL I/O procedures:

```
PROGRAM UPDATE (Newdat, Branch);
VAR
  Newdat : FILE OF INTEGER:
  Branch : FILE OF INTEGER;
               (* main program *)
RESET (Newdat);
OPEN (Branch,
      'NASHUA"PLUGH XYZZY":: MASTER.DAT'.
     HISTORY := NEW) :
REWRITE (Branch):
WHILE NOT EOF (Newdat) DO
  BEGIN
  Branch := Newdat;
  GET (Newdat);
  PUT (Branch);
  END;
CLOSE (Branch);
CLOSE (Newdat);
END.
                (* main program *)
```

This example writes records in a remote file at the node NASHUA. It reads data from a local file known by the logical name NEWDAT and writes the data across the network to the remote file MASTER.DAT in the default device and directory of user PLUGH with password XYZZY.

If you use logical names in your program, you can equate them with either local or remote files. Thus, if your program normally accesses a remote file, and the remote node becomes unavailable, you can bring the volume set containing the file to the local site. You can then mount the volume set and assign the appropriate logical name. For example:

## Remote Access

```
ASSIGN REM::DISK$APPLIC_SET:file-name LOGIC
```

## Local Access

```
$ MOUNT device-name DISK$APPLIC_SET
$ ASSIGN DISK$APPLIC_SET:file-name LOGIC
```

The MOUNT and ASSIGN commands are described in detail in the VAX/VMS DCL Dictionary.

DECnet capabilities are described in the Guide to Networking on VAX/VMS.

# **Error Processing and Condition Handlers**

During the execution of a VAX PASCAL program, errors and exception conditions can occur. They can result from errors during I/O operations, invalid input data, incorrect calls to library routines, arithmetic errors, or system-detected errors. VAX PASCAL provides two methods of error control and recovery:

- 1. VAX Run-Time Library default error-processing procedures
- 2. VAX Condition Handling Facility (including user-written condition handlers)

These error-processing methods are complementary and can be used in the same program.

The VAX Run-Time Library can provide all the condition handling a program needs. By default, it generates error messages for all errors and exception conditions that occur during program execution. Section 5.1 describes how the Run-Time Library processes VAX PASCAL errors.

The VAX Condition Handling Facility provides, at the lowest level, all condition handling for the Run-Time Library. It can also accommodate user-written condition handlers for processing conditions that occur during program execution. Sections 5.2 through 5.5 present a general discussion of condition handling, describe how to write a condition handler in VAX PASCAL, and show how to handle faults and traps. Examples of condition handling are given in Section 5.6.

The use of condition handlers requires considerable programming experience and should not be undertaken by novice users. You should understand the condition-handling descriptions in the VAX/VMS Run-Time Library Routines Reference Manual, the VAX/VMS System Services Reference Manual, and the Introduction to VAX/VMS System Routines before attempting to write your own condition handler.

# **5.1 Run-Time Library Default Error Processing**

When a run-time error occurs, the Run-Time Library prints a message and terminates program execution. These default actions happen unless your program includes a condition handler.

Run-time errors are reported in the following format:

%PAS-x-code, text

The x is a 1-letter abbreviation indicating the severity of the error (see Section 5.4.4). The code is an abbreviation of the error message described in the text. VAX PASCAL run-time errors and recovery procedures are described in Appendix A, Diagnostic Messages. Most Run-Time Library routines provide their own error messages, as described in the VAX/VMS Run-Time Library Routines Reference Manual.

## 5.2 Definitions of Terms

The following terms are used in the discussion of condition handling:

- Condition handler—A function specified by a routine as the handler to be called when an exception condition is signaled.
- Condition value—An integer value that identifies a specific condition.
- Stack frame—A standard data structure built on the stack during a routine call, starting from the location addressed by the Frame Pointer (FP) and proceeding to both higher and lower addresses; it is popped off the stack during the return from a routine.
- Routine activation—The environment in which a routine executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the routine activation. A new routine activation is created every time a routine is called and is deleted when control passes from the routine.
- Establish—The process of placing the address of a condition handler in the stack frame of the current routine activation. A condition handler established for a routine activation is automatically called when a condition occurs. In VAX PASCAL, condition handlers are established by means of the predeclared procedure ESTABLISH. A routine that establishes a condition handler is known as an establisher.
- Program exit status—The status of the program at its completion.

- Signal—The means by which the occurrence of an exception condition is made known. Signals are generated by the operating system in response to I/O events and hardware errors, by the system-supplied library routines, and by user routines. All signals are initiated by a call to the signaling facility, for which there are two entry points:
  - LIB\$SIGNAL—Used to signal a condition and, possibly, to continue program execution
  - LIB\$STOP—Used to signal a severe error and discontinue program execution, unless a condition handler performs an unwind operation
- Resignal—The means by which a condition handler indicates that the signaling facility is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the value SS\$\_RESIGNAL.
- Unwind—The return of control to a particular routine activation, bypassing any intermediate routine activations. For example, if X calls Y, and Y calls Z, and Z detects an error, then a condition handler associated with X or Y can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

# **5.3 Overview of VAX Condition Handling**

When the VAX/VMS system creates a user process, a system-defined condition handler is established in the absence of any user-written condition handler. The system-defined handler processes errors that occur during execution of the user image. Thus, by default, a run-time error causes the system-defined condition handler to print error messages and to terminate or continue execution of the image, depending on the severity of the error.

When a condition is signaled, the system searches for condition handlers to process the condition. The system conducts the search for condition handlers by proceeding down the stack, frame by frame, until a condition handler is found that does not resignal. The default handler calls the system's message output routine to send the appropriate message to the user. Messages are sent to the SYS\$OUTPUT and SYS\$ERROR files. If the condition is not a severe error, program execution continues. If the condition is a severe error, the default handler forces program termination, and the condition value becomes the program exit status.

You can create and establish your own condition handlers according to the needs of your application. For example, a condition handler could create and display messages that describe specific conditions encountered during the execution of your program, instead of relying on system error messages.

## **5.3.1 Condition Signals**

A condition signal consists of a call to either LIB\$SIGNAL or LIB\$STOP, the two entry points to the signaling facility. These entry points must be declared as external procedures, for example:

```
[EXTERNAL, ASYNCHRONOUS]
PROCEDURE LIB$SIGNAL
(%IMMED Condition : INTEGER;
%IMMED FAO_Params : [UNSAFE,LIST] INTEGER);
EXTERN;

[EXTERNAL, ASYNCHRONOUS]
PROCEDURE LIB$STOP
(%IMMED Condition : INTEGER;
%IMMED FAO_Params : [UNSAFE,LIST] INTEGER);
EXTERN;
```

Additional parameters can be included to provide supplementary information about the condition.

If a condition occurs in a routine that is not prepared to handle it, a signal is issued to notify other active routines. If the current routine can continue after the signal is propagated, you can call LIB\$SIGNAL. A higher-level routine can then determine whether program execution should continue. If the nature of the condition does not allow the current routine to continue, you can call LIB\$STOP.

# **5.3.2 Handler Responses**

A condition handler responds to an exception condition by taking action in three major areas:

- 1. Condition correction
- 2. Condition reporting
- 3. Execution control

The handler first determines whether the condition can be corrected. If so, it takes the appropriate action, and execution continues. If the handler cannot correct the condition, the condition may be resignaled; that is, the handler requests that another condition handler be sought to process the condition.

A handler's condition reporting can involve one or more of the following actions:

- Maintaining a count of exceptions encountered during program execu-
- Resignaling the same condition to send the appropriate message to the output file
- Changing the severity field of the condition value and resignaling the condition
- Signaling a different condition; for example, the production of a message designed for a specific application

A handler can control execution in several ways:

- By continuing from the signal. If the signal was issued through a call to LIB\$STOP, the program exits.
- By doing a non-local goto (see example 5 in Section 5.6).
- By unwinding to the establisher at the point of the call that resulted in the exception. The handler can then determine the function value returned by the called routine.
- By unwinding to the establisher's caller (the routine that called the routine which established the handler). The handler can then determine the function value returned by the called routine.

# **5.4 Writing Condition Handlers**

The following sections describe how to write and establish condition handlers and provide some simple examples. See the Introduction to VAX/VMS System Routines, the VAX/VMS Run-Time Library Routines Reference Manual, and the VAX/VMS System Services Reference Manual for more details on condition handlers.

# 5.4.1 Establishing and Removing Handlers

To use a condition handler, you must first declare the handler as a routine in the declaration section of your program; then, within the executable section, you must call the predeclared procedure ESTABLISH. The ESTABLISH procedure sets up a VAX PASCAL language-specific condition handler that in turn allows your handler to be called. User-written condition handlers set up by ESTABLISH must have the ASYNCHRONOUS attribute. Such routines may access only local, READONLY, and VOLATILE variables, and local, predeclared, and ASYNCHRONOUS routines.

Because condition handlers are ASYNCHRONOUS, any attempt to access a non-READONLY or non-VOLATILE variable declared in an enclosing block will result in a warning message. The predeclared file variables INPUT and OUTPUT are such non-VOLATILE variables; therefore, simultaneous access to these files from both an ordinary program and from an ASYNCHRONOUS condition handler's activation may have undefined results. The following steps outline the recommended method for performing I/O operations from a condition handler:

- 1. Declare a file with the VOLATILE attribute at program level.
- 2. Open this file to refer to SYS\$INPUT, SYS\$OUTPUT, or another appropriate file.
- Use this file in the condition handler.

External routines (including system services) that are called by a condition handler require the ASYNCHRONOUS attribute in their declaration.

You should set up a user-written condition handler with the predeclared procedure ESTABLISH rather than with the Run-Time Library routine LIB\$ESTABLISH. ESTABLISH follows the VAX PASCAL procedurecalling rules and is able to handle VAX PASCAL condition handlers more efficiently than LIB\$ESTABLISH. A condition handler set up by LIB\$ESTABLISH might interfere with the default error handling of the PASCAL run-time system, and cause unpredictable results.

To establish a condition handler with either LIB\$ESTABLISH or ESTABLISH, your routine must have the ASYNCHRONOUS attribute. It may access only static, VOLATILE or READONLY variables, ASYNCHRONOUS routines declared at the outermost level of the program, and predeclared PASCAL routines.

The following example shows how to establish a condition handler using the VAX PASCAL procedure ESTABLISH:

```
[EXTERNAL, ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs : Sigarr;
   VAR Mechargs : Mecharr)
                            : INTEGER:
   EXTERN:
ESTABLISH (Handler):
```

To establish the handler, call the ESTABLISH procedure. To remove an established handler, call the predeclared procedure REVERT, as follows:

## REVERT:

As a result of this call, the condition handler established in the current stack frame is removed. When control passes from a routine, any condition handler established during the routine's activation is automatically removed.

## **5.4.2** Parameters for Condition Handlers

A VAX PASCAL condition handler is an integer-valued function that is called when a condition is signaled. Two formal VAR parameters must be declared for a condition handler:

- An integer array to refer to the parameter list from the call to the signal routine (the signal array); that is, the list of parameters included in calls to LIB\$SIGNAL or LIB\$STOP (see Section 5.3.1)
- An integer array to refer to information concerning the routine activation that established the condition handler (the mechanism array)

For example, a condition handler may be defined as follows:

```
TYPE
```

Sigarr = ARRAY[0..9] OF INTEGER; Mecharr = ARRAY[0..4] OF INTEGER;

[EXTERNAL, ASYNCHRONOUS] FUNCTION Handler

(VAR Sigargs : Sigarr;

VAR Mechangs : Mechange : INTEGER;

EXTERN;

ESTABLISH (Handler);

The signal procedure passes the values listed below to the array Sigargs.

Value	Meaning	
Sigargs[0]	The number of parameters being passed in this array (parameter count).	
Sigargs[1]	The primary condition being signaled (condition value). See Section 5.4.4 for a discussion of condition values.	
Sigargs[2 to n]	The optional parameters supplied in the call to LIB\$SIGNAL or LIB\$STOP; note that the index range of Sigargs should include as many entries as are needed to refer to the optional parameters.	

The routine that established the condition handler passes the values listed below to the array Mechargs. These values contain information about the establisher's routine activation.

Value	Meaning	
Mechargs[0]	The number of parameters being passed in this array	
Mechargs[1]	The address of the stack frame that established the handler	
Mechargs[2]	The number of calls that have been made (that is, the stack frame depth) from the routine activation u to the point at which the condition was signaled	
Mechargs[3]	The value of register R0 at the time of the signal	
Mechargs[4]	The value of register R1 at the time of the signal	

## **5.4.3 Handler Function Return Values**

Condition handlers are functions that return values to control subsequent execution. These values and their effects are listed below.

Value	Effect
SS\$_CONTINUE	Continues execution from the signal. If the signal was issued by a call to LIB\$STOP, the program does not continue, but exits.
SS\$_RESIGNAL	Resignals to continue the search for a condition handler to process the condition.

In addition, a condition handler can request a stack unwind by calling SYS\$UNWIND before returning. Declare SYS\$UNWIND as follows:

```
[ASYNCHRONOUS,EXTERNAL(SYS$UNWIND)] FUNCTION $UNWIND (DEPADR : INTEGER := %IMMED O; NEWPC : INTEGER := %IMMED O) : INTEGER; EXTERNAL:
```

When SYS\$UNWIND is called, the function return value is ignored. The handler modifies the saved registers R0 and R1 in the mechanism parameters to specify the called function's return value.

A stack unwind is usually made to one of two places:

• The point in the establisher at which the call was made that resulted in the exception. Specify:

```
Status := $UNWIND (Mechargs[2],0);
```

• The routine that called the establisher. Specify:

```
Status := $UNWIND (Mechargs[2]+1,0);
```

# **5.4.4 Condition Values and Symbols**

VAX/VMS uses condition values to indicate that a called routine has either executed successfully or failed, and to report exception conditions. Condition values are usually symbolic names that represent 32-bit packed records, consisting of fields (usually interpreted as integers) that indicate which system component generated the value, the reason the value was generated, and the severity of the condition. The definition of a condition value can have the form:

TYPE Condition\_Value = PACKED RECORD

(* Field	Bits	Meaning *)
Severity:[POS(0),BIT] 07;	(* 02	One of the following severity codes:  O - warning  1 - success  2 - error  3 - information  4 - severe error  5,6,7 - reserved *)
Message: [POS(3),BIT(13)]08191;	(* 315	The condition that occurred. When bit 15 is 1, it indicates that the message is specific to a single facility. When bit 15 is 0, it indicates a system-wide message. *)
Facility: [POS(16),BIT(12)]04095;	(*1627	The software component that generated the condition value. When bit 27 is 1, it indicates a customer facility. When bit 27 is 0, it indicates a DIGITAL facility. *)
Control: [POS(28),BIT(4)]015; END;	(*2831	Control bits. *)

A warning severity code (0) indicates that although output was produced, the results may be unpredictable. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results may not be correct. A severe error code (4) indicates that the error was of such severity that no output was produced.

A condition handler can alter the severity code of a condition value to allow execution to continue or to force an exit, depending on the circumstances.

Occasionally a condition handler may require a particular condition to be identified by an exact match; that is, each bit of the condition value bits (0..31) must match the specified condition. For example, you may want to process a floating overflow condition only if the severity code is still 4 (that is, if no previous condition handler has changed the severity code) and the control bits have not been modified. A typical condition handler response is to change the severity code and resignal.

In most cases, however, you want some response to a condition, regardless of the value of the severity code or control bits. To ignore the severity and control fields of a condition value, declare and call the LIB\$MATCH\_ COND function (for an example, see the declaration section in Section 5.6).

# **5.5 Handling Faults and Traps**

If a VAX processor detects an error while executing a machine instruction, it can take one of two actions. The first action, called a fault, preserves the contents of registers and memory in a consistent state so that the instruction can be restarted. The second action, called a trap, completes the instruction, but with a predefined result. For example, if an integer overflow trap occurs, the result is the correct low-order part of the true value.

The action taken when an exception occurs depends upon the type of exception. For example, faults occur for access violations and for detection of a floating reserved operand. Traps occur for integer overflow and for integer divide-by-zero exceptions. However, when a floating overflow, floating underflow, or floating divide-by-zero exception occurs, the action taken depends upon the type of VAX processor executing the instruction. The original VAX-11/780 processor traps when these errors occur and stores a floating reserved operand in the destination. All other VAX processors fault on these exceptions, allowing the error to be corrected and the instruction restarted.

If your program is written to handle floating traps, but runs on a VAX processor that generates faults, execution may continue incorrectly. For example, if a condition handler merely causes execution to continue after a floating trap, a reserved operand is stored and the next instruction is executed. However, the same handler used on a processor that generates faults causes an infinite loop of faults because it restarts the erroneous instruction. Therefore, you should write floating-point exception handlers that take the appropriate actions for both faults and traps.

Separate sets of condition values are signaled by the processor for faults and traps. Exceptions and their condition code names are:

Exception	Fault	Trap	
Floating overflow	SS\$_FLTOVF_F	SS\$_FLTOVF	
Floating underflow	SS\$_FLTUND_F	SS\$_FLTUND	
Floating divide-by-zero	SS\$_FLTDIV_F	SS\$_FLTDIV	

To convert faults to traps, you can use the Run-Time Library LIB\$SIM\_ TRAP procedure either as a condition handler or as a called routine from a user-written handler. When LIB\$SIM\_TRAP recognizes a floating fault, it simulates the instruction completion as if a floating trap had occurred. For information on how to use this procedure, refer to the VAX/VMS Run-Time Library Routines Reference Manual.

## 5.6 **Examples of Condition Handlers**

The following declarations are used in the examples in this section.

```
[ASYNCHRONOUS.EXTERNAL(SYS$UNWIND)] FUNCTION $UNWIND
  (DEPADR : INTEGER := %IMMED O;
  NEWPC
         : INTEGER := %IMMED O)
                                    : INTEGER;
  EXTERNAL;
[EXTERNAL, ASYNCHRONOUS] FUNCTION LIB$MATCH_COND
  (Cond_Value : [UNSAFE] INTEGER;
  Cond_Vals
             : [UNSAFE, LIST] INTEGER)
                                           : INTEGER;
  EXTERNAL;
  Sig_Args = ARRAY[0..100] OF INTEGER;
                                                (Signal parameters)
  Mech_Args = ARRAY[0..4] OF [UNSAFE] INTEGER; {Mechanism parameters}
  Cond_Status = [VOLATILE] RECORD
                                                {Condition values}
       CASE INTEGER OF
            O: (STS$_SUCCESS : [POS(O),BIT(1)] BOOLEAN);
            1:(STS$_SEVERITY : [POS(0),BIT(3)] 0..2**3-1);
            2: (STS$_CODE
                              : [POS(3),BIT(12)] 0..2**12-1);
            3: (STS$_FAC_SP
                            : [POS(15),BIT(1)] BOOLEAN);
            4: (STS$_MSG_NO : [POS(3),BIT(13)] 0..2**13-1);
            5: (STS$_CUST_DEF : [POS(27),BIT(1)] BOOLEAN);
            6: (SYS$_FAC_NO : [POS(16),BIT(12)] 0..2**12-1);
            7:(STS$_COND_ID : [POS(3),BIT(25)] 0..2**25-1);
            8:(STS$_INHIB_MSG : [POS(28),BIT(1)] BOOLEAN);
            END;
```

#### **Examples**

```
    [ASYNCHRONOUS] FUNCTION Handler_0

      (VAR SA : Sig_Args;
      VAR MA : Mech_Args)
                             : [UNSAFE] INTEGER;
      IF LIB$MATCH_COND (SA[1], condition-name ,...) <> 0
      THEN
         BEGIN
                                     { do something appropriate }
         Handler_0 := SS$_CONTINUE; { condition handled,
                                       propagate no further }
         END
      ELSE
         Handler_0 := SS$_RESIGNAL; { propagate condition
                                       status to other handlers }
```

This example shows a simple condition handler. The handler identifies the condition being signaled as one that it is prepared to handle and then takes appropriate action. Note that for all unidentified condition statuses, the handler resignals. A handler must always follow this behavior.

```
[ASYNCHRONOUS] FUNCTION Handler_1
      (VAR SA : Sig_Args;
       VAR MA : Mech_Args) : [UNSAFE] INTEGER;
      IF SA[1] = SS$_UNWIND
      THEN
         BEGIN
                                        { cleanup }
         END:
      Handler_1 := SS$_RESIGNAL;
```

When writing a handler, remember that it can be activated with a condition of SS\$\_UNWIND, signifying that the establisher's stack frame is about to be unwound. If the establisher has special cleanup to perform, such as freeing dynamic memory, closing files, or releasing locks, the handler should check for the SS\$\_UNWIND condition status. If there is no cleanup, the required action of resignaling all unidentified conditions results in the correct behavior. On return from a handler activated with SS\$\_UNWIND, the stack frame of the routine that established the handler is deleted ("unwound").

```
3. [ASYNCHRONOUS] FUNCTION Handler_2
      (VAR SA : Sig_Args;
                            : [UNSAFE] INTEGER;
       VAR MA : Mech_Args)
      IF LIB$MATCH_COND (SA[1], condition-name ,...) <> 0
         BEGIN
                                     { cleanup }
                                     { establish function result
         MA[3] := expression;
                                       seen by caller }
        SUNWIND:
                                     { unwind to caller
                                       of establisher }
        END:
      Handler_2 := SS$_RESIGNAL;
      END:
```

A handler may perform a default unwind to force return to the caller of its establisher. If the establisher is a function whose result is expected in R0 or R0 and R1, the handler must establish the return value by modifying the third or third and fourth positions of the mechanism array (the locations of the return R0 and R1 values). If the establisher is a function whose result is returned by the extra-parameter method, the handler must establish the condition value by assignment to the function identifier. In this case, you must observe two additional restrictions:

- The handler must be nested within the function
- The function result must be declared VOLATILE

```
4. [ASYNCHRONOUS] FUNCTION Handler_3
       (VAR SA : Sig_Args;
                             : [UNSAFE] INTEGER;
       VAR MA : Mech_Args)
                                                     ,...) \Leftrightarrow 0
      IF LIB$MATCH_COND (SA[1], condition-name
      THEN
         BEGIN
                                       { cleanup }
                                      { establish function result
         MA[3] := expression;
                                        seen by caller }
                                      { unwind to establisher }
         $UNWIND (MA[2]);
      Handler_3 := SS$_RESIGNAL;
      END;
```

A handler may also force return to its establisher immediately following the point of call. In this case, you should make sure that the handler understands whether the currently uncompleted call was a function call (in which case a returned value is expected) or a procedure call. If the uncompleted call is a function call that will return a value in R0 or R0 and R1, then the handler can modify the mechanism array to supply a value. But if the uncompleted call is a function call that will return a value using the extra-parameter mechanism, then there is no way for the handler to supply a value.

```
5. [ASYNCHRONOUS] FUNCTION Handler_4
    (VAR SA : Sig_Args;
    VAR MA : Mech_Args) : [UNSAFE] INTEGER;

BEGIN
    IF LIB$MATCH_COND (SA[1], condition-name ,...) <> 0
    THEN
        GOTO 99;
    Handler_4 := SS$_RESIGNAL;
    END.
```

A handler may force control to resume at an arbitrary label in its scope. Note that this reference is to a label in an enclosing block, since a GOTO to a local label would simply remain within the handler. In accordance with the VAX Procedure Calling Standard, VAX PASCAL implements references to labels in enclosing blocks by signaling SS\$\_UNWIND in all stack frames that must be deleted.

```
6. FUNCTION EXP_With_Status
      (X : REAL;
       VAR Status : Cond_Status)
                                    : REAL;
      FUNCTION MTH$EXP
         (A : REAL)
            : REAL:
         EXTERNAL:
      [ASYNCHRONOUS] FUNCTION Math_Error
         (VAR SA : Sig_Args;
          VAR MA : Mech_Args)
                                    : [UNSAFE] INTEGER;
         BEGIN { Math_Error }
         IF LIB$MATCH_COND (SA[1], MTH$_FLOOVEMAT, MTH$_FLOUNDMAT) <> 0
         THEN
            BEGIN
            IF Status.STS$_SUCCESS
                                     { record condition status
                                       if no previous error }
                Status := SA[1]::Cond_Status;
                Math_Error := SS$_CONTINUE; { condition handled,
                                              propagate no further }
            END
          ELSE
             Math_Error := SS$_RESIGNAL; { propagate condition status
                                            to other handlers }
             END:
      BEGIN
             { EXP_With_Status }
      STATUS := SS$_SUCCESS;
      ESTABLISH (Math_Error);
      EXP_With_Status := MTH$EXP (X);
      END;
```

This example shows a handler that records the condition status if a floating overflow or underflow error is detected during the execution of the mathematical function MTH\$EXP.

# **Diagnostic Messages**

This appendix summarizes the error messages that can be generated by a VAX PASCAL program at compile time and at run-time.

## **A.1 Compiler Diagnostics**

The VAX PASCAL compiler reports compile-time diagnostics in the source listing (if one is being generated) and summarizes them on the terminal (in interactive mode) or in the batch log file (in batch mode). Compile-time diagnostics are preceded by the following:

- I indicates an informational message that either flags VAX extensions to the PASCAL standard or provides additional Information about a more severe error.
- W indicates a warning that flags an error that may cause unexpected results, but which will not prevent the program from linking and executing.
- E indicates an error that will prevent code from being generated; however, an object module is produced to indicate that E-level messages were detected in the source program.
- F indicates a fatal error.

If the source program contains either E- or F-level messages, the errors must be corrected before the program can be linked and executed.

All diagnostic messages have explanatory text. These messages, and their severity levels are listed as follows. For those messages that are not selfexplanatory, a brief explanation of the error that was detected in your program is provided. Quotation marks (" ") in message text enclose items for which the compiler will replace with the name of a data object when it generates the message.

ABSALIGNCON, Absolute address / alignment conflict

**Error.** The address specified by the AT attribute does not have the number of low-order bits implied by the specified alignment attribute.

ACCMETHCON, Specified ACCESS\_METHOD conflicts with file's record organization

Warning. You cannot specify ACCESS\_METHOD:=DIRECT for a file that has indexed organization or sequential organization and variable-length records. You cannot specify ACCESS\_ METHOD:=KEYED for a file with sequential or relative organization.

ACTHASNOFRML, Actual parameter has no corresponding formal parameter

**Error.** The number of actual parameters specified in a routine call exceeds the number of formal parameters in the routine's declaration, and the last formal parameter does not have the LIST attribute.

ACTMULTPL, Actual parameter specified more than once

**Error.** Each formal parameter (except one with the LIST attribute) can have only one corresponding actual parameter.

ACTPASCNVTMP, Conversion: actual passed is resulting temporary ACTPASRDTMP, Formal requires read access: actual parameter is resulting temporary

ACTPASSIZTMP, Size mismatch: actual passed is resulting temporary ACTPASWRTMP, Formal requires write access: actual parameter is resulting temporary

**Warning.** A temporary variable is created if an actual parameter does not have the size, type, and accessibility properties required by the corresponding foreign formal parameter.

ACTPRMORD, Actual parameter must be ordinal

**Error.** The actual parameter that specifies the starting index of an array for the PACK or UNPACK procedure must have an ordinal type.

ADDIWRDALIGN, ADD\_INTERLOCKED requires variable with at least word alignment

ADDIWRDSIZE, ADD\_INTERLOCKED requires 16-bit variable

**Error.** These restrictions are imposed by the VAX ADAWI instruction

ADDRESSVAR, "parameter name" is a VAR parameter, ADDRESS is illegal

**Warning.** You should not use the ADDRESS function on a non-VOLATILE variable or component or on a formal VAR parameter.

ALIGNAUTO, Alignment greater than 2 conflicts with automatic allocation

**Error.** The VAX hardware aligns the stack on a longword boundary; therefore, you cannot specify a greater alignment for automatically allocated variables.

ALIGNFNCRES, Alignment greater than 2 not allowed on function result

**Error.** The use of an attribute on a routine conflicts with the requirements of the object's type.

ALIGNINT, ALIGNED expression must be INTEGER value in range 0..9

Error.

ALIGNVALPRM, Alignment greater than 2 not allowed on value parameter

**Error.** The use of an attribute on a parameter conflicts with the requirements of the object's type.

ALLPRMSAM, All parameters to 'MIN' or 'MAX' must have the same type

Error.

APARMACTDEF, Anonymous parameter "parameter number" has neither actual nor default

Error. If the declaration of a routine failed to specify a name for a formal parameter, a call to the routine will result in this error message. Note that the routine declaration would also cause an error to be reported.

ARITHOPNDREQ, Arithmetic operand(s) required

Error.

ARRCNTPCK, Array cannot be PACKED

**Error.** At least one parameter to the PACK or UNPACK procedure must be unpacked.

ARRHAVSIZ, "routine name" requires that ARRAY component have compile-time known size

Error. You cannot use the PACK and UNPACK procedures to pack or unpack one multidimensional conformant array into another. The component type of the dimension being copied must have a compile-time known size; that is, it must have some type other than a conformant schema.

ARRMSTPCK, Array must be PACKED

**Error.** At least one parameter to the PACK or UNPACK procedure must be PACKED.

ARRNOTSTR, Array type is not a string type

Error. You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. The only legal ARRAY, therefore, is a PACKED ARRAY [1..n] OF CHAR.

ASYREQASY, ASYNCHRONOUS "calling routine" requires that "called routine" also be ASYNCHRONOUS

Warning.

ASYREQVOL, ASYNCHRONOUS "routine name" requires that "variable name" be VOLATILE

**Warning.** A variable referred to in a nested ASYNCHRONOUS routine must have the VOLATILE attribute.

ATINTUNS, AT address must be an INTEGER or UNSIGNED value Error.

ATREXTERN, "attribute name" attribute allowed only on external routines

**Error.** The LIST and CLASS\_S attributes can be specified only with the declarations of external routines.

ATTRCONCMDLNE, Attribute contradicts command line qualifier

**Error.** The double-precision attribute specified contradicts the /G\_FLOATING or /NOG\_FLOATING qualifier specified with the PASCAL command.

ATTRCONFLICT, Attribute conflict: "attribute name"

**Information.** This message can appear as additional information on other error messages.

ATTRONTYP, Descriptor class attribute not allowed on this type

**Error.** The use of the descriptor class attribute on the variable, parameter, or routine conflicts with the requirements of the object's type.

AUTOGTRMAXINT, Allocation of "variable name" causes automatic storage to exceed MAXINT bits

**Error.** The VAX implementation restricts automatic storage to a size of 2,147,483,647 bits.

BADSETCMP, < and > not permitted in set comparisons Error.

BINOCTHEX, Expecting BIN, OCT, or HEX

**Error.** You must supply BIN, OCT, or HEX as a variable modifier when reading the variable on a non-decimal basis.

BLKNOTFND, "routine" block "routine name" declared FORWARD in "block name" is missing

Error.

BLKTOODEEP, Routine blocks nested too deeply

Error. You cannot nest more than 31 routine blocks.

BNDACTDIFF, Actual's array bounds differ from those of other parameters in same section

**Error.** All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible.

BNDCNFRUN, Bounds of conformant ARRAY "array name" not known until run-time

Error. You cannot use the UPPER and LOWER functions on a dynamic array parameter in a compile-time constant expression.

BNDSUBORD, Bound expressions in a subrange type must be ordinal

**Error.** The expressions that designate the upper and lower limits of a subrange must be of an ordinal type.

BOOLOPREQ, BOOLEAN operand(s) required

**Error.** The operation being performed requires operands of type BOOLEAN. Such operations include the AND, OR, and NOT operators and the SET\_INTERLOCKED and CLEAR\_INTERLOCKED functions.

BOOSETREQ, BOOLEAN or SET operand(s) required Error.

BYTEALIGN, Type larger than 32 bits can be positioned only on a byte boundary

**Error.** See Chapter 1, VAX PASCAL System Environment, for information on the types that are allocated more than 32 bits.

CARCONMNGLS, CARRIAGE\_CONTROL parameter is meaningless given file's type

**Warning.** The carriage-control parameter is usually meaningful only for files of type TEXT and VARYING OF CHAR.

CASLABEXPR, Case label and case selector expressions are not compatible

**Error.** All case labels in a CASE statement must be compatible with the expression specified as the case selector.

CASORDRELPTR, Compile-time cast allowed only between ordinal, real, and pointer types

CASSELORD, Case selector expression must be an ordinal type Error.

CASSRCSIZ, Source type of a cast must have a size known at compile-time

CASTARSIZ, Target type of a cast must have a size known at compile-time

Error. A variable being cast by the type cast operator cannot be a conformant array or a conformant VARYING parameter. An expression being cast cannot be a conformant array parameter, a conformant VARYING parameter, or a VARYING OF CHAR expression. The target type of the cast cannot be VARYING OF CHAR.

CDDABORT, %DICTIONARY processing of CDD record definition aborted

**Error.** The PASCAL compiler is unable to process the CDD record description. See the accompanying messages for more information.

CDDBADDIR, %DICTIONARY directive not allowed in deepest %INCLUDE, ignored

**Error.** A program may not use the %DICTIONARY directive in the fifth nested %INCLUDE level. The compiler ignores all %DICTIONARY directives in the fifth nested %INCLUDE level.

CDDBADPTR, invalid pointer was specified in CDD record description Warning. The CDD pointer datatype referred to a CDD pathname which cannot be extracted and is replaced by 'INTEGER.

CDDBIT, Ignoring bit field in CDD record description

Information. The PASCAL compiler cannot translate a CDD bit datatype which is not aligned on a byte boundary and whose size is greater than 32 bits.

CDDBLNKZERO, Ignoring blank when zero attribute specified in CDD record description

Information. The PASCAL compiler does not support the CDD BLANK WHEN ZERO clause.

CDDCOLMAJOR, CDD description specifies a column-major array Error. The PASCAL compiler only supports row-major arrays. Change the CDD description to specify a row-major array.

CDDDEPITEM, Ignoring depends item attribute specified in CDD record description

Information. The PASCAL compiler does not support the CDD DEPENDING ON ITEM attribute.

CDDDFLOAT, D\_Floating CDD datatype was specified when compiling with G\_FLOATING

Warning. The CDD record description contains a D\_Floating datatype while compiling with G\_Floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDFLDVAR, CDD record description contains field(s) after CDD variant clause

Error. The CDD record description contains fields after the CDD variant clause. Since PASCAL translates a CDD variant clause into a PASCAL variant clause, and a PASCAL variant clause must be the last field in a record type definition, the fields following the CDD variant clause are illegal.

CDDGFLOAT, G\_Floating CDD datatype was specified when compiling with NOG\_FLOATING

**Warning.** The CDD record description contains a G\_Floating datatype while compiling with D\_Floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDILLARR, Aligned array elements can not be represented, replacing with [BIT(n)] RECORD END

**Information.** The PASCAL compiler does not support CDD record descriptions which specify an array whose array elements are aligned on a boundary greater than the size needed to represent the datatype. It is replaced with [BIT(n)] RECORD END, where 'n' is the appropriate length in bits.

CDDINITVAL, Ignoring specified initial value specified in CDD record description

**Information.** The PASCAL compiler does not support the CDD INITIAL VALUE clause.

CDDMINOCC, Ignoring minimum occurs attribute specified in CDD record description

**Information.** The PASCAL compiler does not support the CDD MINIMUM OCCURS attribute.

CDDONLYTYP, %DICTIONARY may only appear in a TYPE definition part

**Error.** The %DICTIONARY directive is only allowed in the TYPE section of a program.

CDDRGHTJUST, Ignoring right justified attribute specified in CDD record description

**Information.** The PASCAL compiler does not support the CDD JUSTIFIED RIGHT clause.

CDDSCALE, Ignoring scaled attribute specified in CDD record description

**Information.** The PASCAL compiler does not support the CDD scaled datatypes.

CDDSRCTYPE, Ignoring source type attribute specified in CDD record description

**Information.** The PASCAL compiler does not support the CDD source type attribute.

CDDTAGDEEP, CDD description nested variants too deep

**Error.** A CDD record description may not include more than fifteen levels of CDD variants. The compiler ignores variants beyond the fifteenth level.

CDDTAGVAR, Ignoring tag variable and any tag values specified in CDD record description

**Information.** The PASCAL compiler does not fully support CDD VARIANTS OF field description statement. The specified tag variable and any tag values are ignored.

CDDTOODEEP, CDD description nested too deep

**Error.** Attributes for the CDD record description exceed implementation's limit for record complexity. Modify the CDD description to reduce the level of nesting in the record description.

CDDTRUNCREF, Reference string which exceeds 255 characters has been truncated

**Information.** The PASCAL compiler does not support reference strings greater than 255 characters. The reference string is truncated.

CDDUNSTYP, Unsupported CDD datatype "standard data type name"

**Information.** The CDD record description for an item has attempted to use a datatype that is not supported by VAX PASCAL. The PASCAL compiler makes the datatype accessible by declaring it as [BYTE(n)] RECORD END where 'n' is the appropriate length in bytes. Change the datatype to one that is supported by PASCAL or manipulate the contents of the field by passing it to external routines as variables or by using the PASCAL type casting capabilities to perform an assignment.

CLSCNFVAL, CLASS\_S is only valid with conformant strings passed by value

**Error.** When the CLASS\_S attribute is used in the declaration of an internal routine, it can only be used on a conformant PACKED ARRAY OF CHAR or a conformant VARYING. The conformant variable must also be passed by value semantics.

CLSNOTALLW, "descriptor class name" not allowed on a parameter of this type

**Error.** Descriptor class attributes are not allowed on formal parameters defined with either an immediate or reference passing mechanism.

CMTBEFEOF, Comment not terminated before end of input

Error.

CNFCANTCNF, Component of PACKED conformant schema cannot be conformant

Error.

CNTBEARRCMP, Not allowed on an array component

CNTBEARRIDX, Not allowed on an array index

CNTBECAST, Not allowed on a cast

CNTBECNFCMP, Not allowed on a conformant array component

CNTBECNFIDX, Not allowed on a conformant array index

CNTBECNFVRY, Not allowed on a conformant varying component

CNTBECOMP, Not allowed on a compilation unit

CNTBECONST, Not allowed on a CONST definition part

CNTBEDEFDECL, Not allowed on any definition or declaration part

CNTBEDESPARM, Not allowed on a %DESCR foreign mechanism

CNTBEEXESEC, Not allowed on an executable section

CNTBEFILCMP, Not allowed on a file component

CNTBEFUNC, Not allowed on a function result

CNTBEIMMPARM, Not allowed on a parameter passed by an immediate passing mechanism

CNTBELABEL, Not allowed on a LABEL declaration part

CNTBEPCKCNF, Not allowed on a PACKED conformant array component

CNTBEPTRBAS, Not allowed on a pointer base CNTBERECFLD, Not allowed on a record field

CNTBEREFPARM, Not allowed on a parameter passed by a reference passing mechanism

CNTBERTNDECL, Not allowed on a routine declaration

CNTBERTNPARM, Not allowed on a routine parameter

CNTBESETRNG, Not allowed on a set range

CNTBESTDPARM, Not allowed on a %STDESCR foreign mechanism parameter

CNTBETAGFLD, Not allowed on a variant tag field

CNTBETAGTYP, Not allowed on a variant tag type

CNTBETYPDEF, Not allowed on a type definition

CNTBETYPE, Not allowed on a TYPE definition part

CNTBEVALPARM, Not allowed on a value parameter

CNTBEVALUE, Not allowed on a VALUE initialization part

CNTBEVALVAR, Not allowed on a VALUE variable

CNTBEVAR, Not allowed on a VAR declaration part

CNTBEVARBLE, Not allowed on a variable

CNTBEVARPARM, Not allowed on a VAR parameter

CNTBEVRYCMP, Not allowed on a varying component

**Information.** These messages can appear as additional information on other error messages.

COMCONFLICT, COMMON "block name" conflicts with another COMMON or PSECT of same name

**Error.** You can allocate only one variable in a particular common block, and the name of the common block cannot be the same as the names of other common blocks or program sections used by your program.

CSTRBADTYP, Constructor: only ARRAY, RECORD, or SET type

CSTRCOMISS, Constructor: component(s) missing

CSTRNOVRNT, Constructor: no matching variant

CSTRREFAARR, Repetition factor allowed only in ARRAY constructors

CSTRREFAINT, Repetition factor must be INTEGER

CSTRREFALRG, Repetition factor too large

CSTRREFANEG, Repetition factor cannot be negative

CSTRTOOMANY, Constructor: too many components

**Error.** You can write constructors only for data items of an ARRAY type. You must specify one and only one value in the constructor for each component of the type. In an array constructor, you cannot use a negative integer value as a repetition factor to specify values for consecutive components.

CTGARRDESC, Contiguous array descriptor cannot describe size /alignment properties

**Information.** Conformant array parameters, dynamic array parameters, and %DESCR array parameters all use the contiguous array descriptor mechanism in the VAX Procedure Calling Standard. Size and alignment attributes are prohibited on such arrays, as these attributes can create noncontiguous allocation. This message can appear as additional information in other error messages.

DEBUGOPT, /NOOPTIMIZE is recommended with /DEBUG

**Information.** Unexpected results may be seen when debugging an optimized program. To prevent conflicts between optimization and debugging, you should compile your program with /NOOPTIMIZE until it is thoroughly debugged. Then you can recompile the program with optimization enabled to produce more efficient code.

DEFVARPARM, Default parameter syntax not allowed on VAR parameters

DEFRTNPARM, Default parameter syntax not allowed on routine parameters

Error.

DESCTYPCON, Descriptor class / type conflict

**Error.** The descriptor class for parameter passing conflicts with the parameter's type. Refer to Chapter 3, Section 3.2 for legal descriptor class/type combinations.

DIRCONVISIB, Directive contradicts visibility attribute

**Error.** The EXTERN, EXTERNAL, and FORTRAN directives conflict directly with the LOCAL and GLOBAL attributes.

DONTPACKVAR, "routine name" is illegal, variable can never appear in a packed context

**Error.** You cannot call the BITSIZE and BITNEXT functions for conformant parameters.

DUPLALIGN, Alignment already specified DUPLALLOC, Allocation already specified DUPLATTR, Attribute already specified DUPLCLASS, Descriptor class already specified DUPLDOUBLE, Double precision already specified

**Error.** Only one member of a particular attribute class can appear in the same attribute list.

DUPLGBLNAM, Duplicated global name

**Warning.** The GLOBAL attribute cannot appear on more than one variable or routine with the same name.

DUPLMECH, Passing mechanism already specified DUPLOPT, Optimization already specified DUPLSIZE, Size already specified DUPLVISIB, Visibility already specified

**Error.** Only one member of a particular attribute class can appear in the same attribute list.

DUPTYPALI, Alignment already specified by type identifier "type name"

DUPTYPALL, Allocation already specified by type identifier "type name"

DUPTYPATR, Attribute already specified by type identifier "type name"

DUPTYPDES, Descriptor class already specified by type identifier "type name"

DUPTYPSIZ, Size already specified by type identifier "type name" DUPTYPVIS, Type identifier "type name" already carries a visibility attribute

**Error.** An attribute specified for an object was already specified in the definition of the object's type.

### ELEOUTRNG, Element out of range

**Error.** A value specified in a set constructor used as a compiletime constant expression does not fall within the subrange defined as the set's base type.

## EMPTYCASE, Empty case body

**Error.** You failed to specify any case labels and corresponding statements in the body of a CASE statement.

ENVERROR, Environment resulted from a compilation with Errors

**Error.** The environment file inherited by the program compiled with errors or warnings. Unexpected results may occur in the program now being compiled. This message is at warning level if the environment file compiled with warnings; it is at error level if the environment file compiled with errors.

ENVOLDVER, Environment was created by a VAX PASCAL V2 compiler, please recompile

**Warning.** The environment file inherited by the program was created by a VAX PASCAL V2 compiler. You should regenerate the environment file with the VAX PASCAL V3 compiler.

ENVFATAL, Environment resulted from a compilation with Fatal Errors

**Error.** The environment file inherited by the program compiled with errors or warnings. Unexpected results may occur in the program now being compiled. This message is at warning level if the environment file compiled with warnings; it is at error level if the environment file compiled with errors.

ENVWARN, Environment resulted from a compilation with Warnings

**Warning.** The environment file inherited by the program compiled with errors or warnings. Unexpected results may occur in the program now being compiled. This message is at warning level if the environment file compiled with warnings; it is at error level if the environment file compiled with errors.

ERREALCNST, Error in real constant: digit expected

Error.

ERRNONPOS, ERROR parameter can be specified only with nonpositional syntax

Error.

ERRORLIMIT, Error Limit = "current error limit", source analysis terminated

**Fatal.** The error limit specified for the program's compilation was exceeded; the compiler was unable to continue processing the program. By default, the error limit is set at 30, but you can use the /ERROR\_LIMIT qualifier at compile time to change it.

ESTBASYNCH, ESTABLISH requires that "routine name" be ASYNCHRONOUS

Warning.

EXPLCONVREQ, Explicit conversion to lower type required

**Error.** An expression of a higher-ranked type cannot be assigned to a variable of a lower-ranked type; you must first convert the higher-ranked expression by using DBLE, SNGL, TRUNC, ROUND, UTRUNC, or UROUND, as appropriate.

EXPRARITH, Expression must be arithmetic

**Error.** An expression whose type is not arithmetic cannot be assigned to a variable of a real type.

EXPRARRIDX, Expression is incompatible with unpacked array's index type

**Error.** The index type of the UNNPACKed array is not compatible with the index type of either the PACK or UNPACK procedure it was passed to.

EXPRCOMTAG, Expression is not compatible with tag type

**Error.** A case label specified for a NEW, DISPOSE, or SIZE routine must be assignment compatible with the tag type of the variant.

EXPRNOTSET, Expression is not a SET type

**Error.** The compiler encountered an expression of some type other than SET in a call to the CARD function.

EXTRNALLOC, Allocation attribute conflicts with EXTERNAL visibility

**Error.** The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

EXTRNAMDIFF, External names are different

**Information.** This message can appear as additional information on other error messages.

EXTRNCFLCT, "PSECT or FORWARD" conflicts with EXTERNAL visibility

**Error.** The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

FILEVALASS, FILE evaluation / assignment is not allowed

**Error.** You cannot attempt to evaluate a file variable or assign values to it.

FILOPNDREQ, FILE operand required

Error. The EOF, EOLN, and UFB functions require parameters of file types.

FILVARFIL, FILE\_VARIABLE parameter must be of a FILE type

Error. The file-variable parameter to the OPEN and CLOSE procedures must denote a file variable.

FLDIVPOS, Field "field name" is illegally positioned

**Error.** A POS attribute attempted to position a record field before the end of the previous field in the declaration.

FLDONLYTXT, Field width allowed only when writing to a TEXT file FLDWDTHINT, Field-width expression must be of type INTEGER

Error.

FORACTORD, FOR loop control variable must be of an ordinal type FORACTVAR, FOR loop control must be a true variable

Error. The control variable of a FOR statement must be a simple variable of an ordinal type and must be declared in a VAR section. For example, it could not be a field in a record that was specified by a WITH statement, or a function identifier.

FORCTLVAR, "variable name" is a FOR control variable

Warning. The control variable of a FOR statement cannot be assigned a value; used as a parameter to the ADDRESS function; passed as a writeable VAR, %REF, %DESCR, or %STDESCR parameter; used as the control variable of a nested FOR statement; or written into by a READ, READLN, or READV procedure.

FORINEXPR, Expression is incompatible with FOR loop control variable

**Error.** The type of the initial or final value specified in a FOR statement is variable.

FRMLPRMDESC, Formal parameters use different descriptor formats FRMLPRMINCMP, Formal routine parameters are not compatible FRMLPRMNAM, Formal parameters have different names FRMLPRMSIZ, Formal parameters have different size attributes FRMLPRMTYP, Formal parameters have different types

**Information.** These messages can appear as additional information on other error messages.

FRSTPRMSTR, READV requires first parameter to be a string expression

**Error.** You must specify at least two parameters for the READV procedure—a character-string expression and a variable into which new values will be read.

FRSTPRMVARY, WRITEV requires first parameter to be a variable of type VARYING

Error.

FUNCTRESTYP, Routine must be declared as FUNCTION to specify a result type

**Error.** You cannot specify a result type on a PROCEDURE declaration.

FUNRESTYP, Function result types are different

**Information.** This message can appear as additional information on other error messages.

FWDREPATRLST, Declared FORWARD; repetition of attribute list not allowed

FWDREPPRMLST, Declared FORWARD; repetition of formal parameter list not allowed

FWDREPRESTYP, Declared FORWARD; repetition of result type not allowed

**Error.** If the heading of a routine has the FORWARD directive, the declaration of the routine body cannot repeat the formal parameter list, the result type (applies only if the routine is a function), nor any attribute lists that appeared in the heading.

FWDWASFUNC, FORWARD declaration was FUNCTION FWDWASPROC, FORWARD declaration was PROCEDURE

Error.

GOTONOTALL, GOTO not allowed to jump into a structured statement

Warning.

GTR32BITS, "routine name" cannot accept parameters larger than 32 bits

Error. DEC or UDEC cannot translate objects larger than 32 bits into their textual equivalent.

HIDATOUTER, HIDDEN legal only on definitions and declarations at outermost level

**Error.** When an environment file is being generated, it is possible to prevent information concerning a declaration from being included in the environment file by using the HIDDEN attribute. However, since an environment file consists of only declarations and definitions at the outermost level of a compilation unit, the HIDDEN attribute is only legal on these definitions and declarations.

IDENTGTR31, Identifier longer than 31 characters exceeds capacity of compiler

Warning.

IDXNOTCOMPAT, Index type is not compatible with declaration

**Error.** The type of an index expression is not assignment compatible with the index type specified in the array's type definition.

IDXREQDKEY, Creating INDEXED organization requires dense keys

**Warning.** When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's alternate keys must be dense; that is, you may not omit any key numbers in the range from 0 through the highest key number specified for the file's component type.

IDXREQKEY0, Creating INDEXED organization requires FILE OF RECORD with at least KEY(0)

**Warning.** When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's component type must be a RECORD for which a primary key, designated by the [KEY(0)] attribute, is defined.

IMMEDBNDROU, Immediate passing mechanism may not be used on bound routine "routine name"

**Warning.** You may not prefix a formal or an actual routine parameter with the immediate passing mechanism unless the routine was declared with the UNBOUND attribute.

IMMEDUNBND, Routines passed by immediate passing mechanism must be UNBOUND

**Warning.** A formal routine parameter that has the immediate passing mechanism must also have the UNBOUND attribute.

IMMGTR32, Immediate passing mechanism not allowed on values larger than 32 bits

**Error.** See Chapter 1, VAX PASCAL System Environment, for more information on the types that are allocated more than 32 bits.

IMMHAVSIZ, Type passed by immediate passing mechanism must have compile-time known size

**Error.** You cannot specify an immediate passing mechanism for a conformant parameter or a formal parameter of type VARYING OF CHAR.

INCMPBASE, Incompatible with SET base type

**Error.** If no type identifier denotes the base type of a set constructor, the first element of the constructor determines the base type. The type of all subsequent elements specified in the constructor must be compatible with the type of the first.

INCMPOPND, Incompatible operand(s)

**Error.** The types of one or more operands in an expression are not compatible with the operation being performed.

INCMPPARM, Incompatible "routine" parameter

**Error.** An actual routine parameter is incompatible with the corresponding formal parameter.

INCMPTAGTYP, Incompatible variant tag types

**Error.** This message can appear as additional information on other error messages.

INCTOODEEP, %INCLUDE directives nested too deeply, ignored

**Error.** A program may not include more than five levels of files with the %INCLUDE directive. The compiler ignores %INCLUDE files beyond the fifth level.

INDNOTORD, Index type must be an ordinal type

Error. The index type of an array must be an ordinal type.

INITNOEXT, INITIALIZE routine may not be EXTERNAL INITNOFRML, INITIALIZE routine must have no formal parameter list

Error.

INPNOTDECL, INPUT not declared in heading

**Error.** A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

IVATTR, Unrecognized attribute

Error.

IVAUTOMOD, AUTOMATIC variable is illegal at the outermost level of a MODULE

**Error.** You may not specify the AUTOMATIC attribute for a variable declared at module level.

IVCHKOPT, Unrecognized CHECK option

Warning.

IVCOMBFLOAT, Illegal combination of D\_floating and G\_floating

**Error.** You cannot combine D\_floating and G\_floating numbers in a binary operation.

IVDIRECTIVE, Unrecognized directive

**Error.** The directive following a PROCEDURE or FUNCTION heading is not one of those recognized by the VAX PASCAL compiler.

IVENVIRON, Environment "environment name" has illegal format, source analysis terminated

**Fatal.** The environment file inherited by the program has an illegal format; compilation is immediately aborted. However, a listing will still be produced if one was being generated.

IVFUNC, Invalid use of function "function name" IVFUNCALL, Invalid use of function call IVFUNCID, Invalid use of function identifier

**Error.** These messages result from illegal attempts to assign values or otherwise refer to the components of the function result (if its type is structured), use the type cast operator on a function identifier or its result, or deallocate the storage reserved for the function result if its type is a pointer.

IVKEYVAL, FINDK KEY\_VALUE cannot be an array (other than PACKED ARRAY [1..n] OF CHAR)

Error.

IVKEYWORD, Missing or unrecognized keyword

**Error.** The compiler failed to find an identifier where it expected one in a call to the OPEN or CLOSE procedure, or it found an identifier that was not legal in this position in the parameter list.

IVMATCHTYP, MATCH\_TYPE parameter to FINDK must be EQL, GEQ or GTR

Error.

IVOPTMOPT, Unrecognized OPTIMIZE option

Warning.

IVQUALFILE, Illegal qualifier "qualifier name" on file specification

**Warning.** Only the /LIST and /NOLIST qualifiers are allowed on the file specification of a %INCLUDE directive.

IVQUOCHAR, Illegal nonprinting character (ASCII "nnn") within quotes

**Warning.** The only nonprinting characters allowed in a quoted string are the space and tab; the use of other nonprinting characters in a string causes this warning. To include nonprinting characters in a string, you should use the extended string syntax described in *Programming in VAX PASCAL*.

IVRADIXDGIT, Illegal digit in binary, octal, or hexadecimal constant **Error.** 

IVREDECL, Illegal redeclaration gives "symbol name" multiple meanings in "scope name"

IVREDECLREC, Illegal redeclaration gives "symbol name" multiple meanings in this record

IVREDEF, Illegal redefinition gives "symbol name" multiple meanings in "scope name"

**Warning.** When an identifier is used in any given block, it must have the same meaning wherever it appears in the block.

IVUSEALIGN, Invalid use of alignment attribute IVUSEALLOC, Invalid use of allocation attribute

Error.

Error.

IVUSEATTR, Invalid use of "attribute name" attribute

**Error.** The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEATTRLST, Invalid use of an attribute list

IVUSEBNDID, Illegal use of bound identifier "identifier name"

**Error.** An identifier that represents one bound of a conformant schema was used where a variable was expected, such as in an assignment statement or in a formal VAR parameter section. The restrictions on the use of a bound identifier are identical to those on a constant identifier.

IVUSEDES, Invalid use of descriptor class attribute

**Error.** The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEFNID, Illegal use of function identifier "identifier name"

**Error.** Two examples of illegal uses are the assignment of values to the components of the function result (if its type is structured) and the passing of the function identifier as a VAR parameter.

IVUSESIZ, Invalid use of size attribute

**Error.** The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

KEYINTRNG, KEY number must be an INTEGER value in range 0..254

**Error.** The key number specified by a KEY attribute must fall in the integer subrange 0..254.

KEYNOTALIGN, KEY "key number" field "field name" at bit position "bit position" is unaligned

KEYORDSTR, KEY allowed only on ordinal and fixed-length string fields

KEYPCKREC, KEY field in PACKED RECORD must have an alignment attribute

KEYREDECL, Key number "key number" is multiply defined KEYSIZ1\_2\_4, Size of an ordinal key must be 1, 2 or 4 bytes KEYSIZ2\_4, Size of a signed integer key must be 2 or 4 bytes KEYSIZSTR, Size of a string key cannot exceed 255 bytes KEYUNALIGN, KEY field cannot be UNALIGNED

Error.

LABDECIMAL, Label number must be expressed in decimal radix **Error.** 

LABINCTAG, Variant case label's type is incompatible with tag type

**Error.** The type of a constant specified as a case label of a variant record is not assignment compatible with the type of the tag field.

LABNOTFND, No definition of label "label name" in statement part of "block name"

Error. A label that you declared in a LABEL section does not prefix a statement in the executable section.

LABREDECL, Redefinition of label "label name" in "block name"

**Error.** A label may not prefix more than one statement in the same block.

LABRNGTAG, Variant case label does not fall within range of tag type

**Error.** A constant specified as a case label of a variant record is not within the range defined for the type of the tag field.

LABTOOBIG, Label "label number" is greater than MAXINT Error.

LABUNDECL, Undeclared label "label name"

**Error.** PASCAL requires that you declare all labels in a LABEL declaration section before you use them in the executable section.

LABUNSATDECL, Unsatisfied declaration of label "label name" is not local to "block name"

**Error.** A label that prefixes a statement in a nested block was declared in an enclosing block.

LIBESTAB, LIB\$ESTABLISH is incompatible with VAX PASCAL; use predeclared procedure ESTABLISH

Warning. VAX PASCAL establishes its own condition handler for processing PASCAL-specific run-time signals. Calling LIB\$ESTABLISH directly replaces the handler supplied by the compiler with a user-written handler; the probable result is improper handling of run-time signals. You should use PASCAL's predeclared ESTABLISH procedure to establish user-written condition handlers.

LISTONEND, LIST attribute allowed only on final formal parameter **Error.** 

LISTUSEARG, Formal parameter has LIST attribute, use predeclared function ARGUMENT

**Error.** A formal parameter with the LIST attribute cannot be directly referenced. You should use the predeclared function ARGUMENT to reference the actual parameters corresponding to the formal parameter.

LNETOOLNG, Line too long, is truncated to 255 characters

**Error.** A source line cannot exceed 255 characters. If it does, the compiler disregards the remainder of the line.

LOWGTRHIGH, Low-bound exceeds high-bound

**Error.** The definition of the flagged subrange type is illegal because the value specified for the lower limit exceeds that for the upper limit.

MAXLENINT, Max-length must be a value of type INTEGER

**Error.** The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXLENRNG, Max-length must be in range 1..65535

**Error.** The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MECHEXTERN, Foreign mechanism specifier allowed only on external routines

Error.

MISSINGEND, No matching END, expected near line "line number"

**Information.** The compiler expected an END at a location where none was found. Compilation proceeds as though the END were correctly located.

MODOFNEGNUM, MOD of a negative modulus has no mathematical definition

**Error.** In the MOD operation A MOD B, the operand B must have a positive integer value. This message is issued only when the MOD operation occurs in a compile-time constant expression.

MSTBEARRAY, Type must be ARRAY

Error.

MSTBEARRVRY, Type must be ARRAY or VARYING

Error. You cannot use the syntax [index] to refer to an object that is not of type ARRAY or VARYING OF CHAR.

MSTBEBOOL, Control expression must be of type BOOLEAN

Error. The IF, REPEAT, and WHILE statements require a Boolean control expression.

MSTBEREC, Type must be RECORD

Error.

MSTBERECVRY, Type must be RECORD or VARYING

**Error.** You cannot use the syntax "variable.identifier" to refer to an object that is not of type RECORD or VARYING OF CHAR.

MSTBESTAT, Cannot initialize non-STATIC variables

Error. You cannot initialize variables declared without the STATIC attribute in nested blocks, nor can you initialize program-level variables whose attributes give them some allocation other than STATIC.

MSTBETEXT, "I/O routine" requires FILE\_VARIABLE of type TEXT

**Error.** The READLN and WRITELN procedures operate only on text files.

MULTDECL, "symbol name" has multiple conflicting declarations, reason(s):

Error.

NCATOA, Cannot reformat content of actual's CLASS\_NCA descriptor as CLASS\_A

**Error.** This message can appear as additional information on other error messages.

NEWQUADAGN, "type name"'s base type is ALIGNED("nnn"); NEW handles at most ALIGNED(3)

**Error.** You cannot call the NEW procedure to allocate pointer variables whose base types specify alignment greater than a quadword. To allocate such variables, you must use external routines.

NOASSTOFNC, Block does not contain an assignment to function result "function name"

**Warning.** The block of a function must include a statement that assigns a return value to the function identifier.

NODECLVAR, "symbol name" is not declared in a VAR section of "block name"

**Error.** You cannot initialize a variable using the VALUE section if the variable was not declared in the same block in which the VALUE section appears.

NODSCREC, No descriptor class for RECORD type

**Error.** The VAX Procedure Calling Standard does not define a descriptor format for records; therefore, you cannot specify %DESCR for a parameter of type RECORD.

NOFLDREC, No field "field name" in RECORD type "type name"

Error. The field specified does not exist in the specified record.

NOFRMINDECL, Declaration of "routine" parameter "routine name" supplied no formal parameter list

**Information.** You specified actual parameters in a call on a formal routine parameter that was declared with no formal parameters. Although such a call was legal in VAX PASCAL Version 1, it does not follow the rules of the Pascal standard. You should edit your program to reflect this change.

NOINITEXT, Initialization not allowed on EXTERNAL variables NOINITINH, Initialization not allowed on inherited variables

**Error.** You can initialize only those variables whose storage is allocated in this compilation.

NOINITVAR, Cannot initialize "symbol name"—it is not declared as a variable

**Error.** Variables are the only data items that can be initialized, and they can be initialized only once.

NOLISTATTR, Parameter to this predeclared function must have LIST attribute

**Error.** ARGUMENT and ARGUMENT\_LIST\_LENGTH require their first parameter to be a formal parameter with the LIST attribute.

NOREPRE, No textual representation for values of this type

**Error.** You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. Such types are RECORD, ARRAY (other than PACKED ARRAY [1..n] OF CHĀR), SET, and pointer.

NOTAFUNC, "symbol name" is not declared as a "routine."

**Error.** An identifier followed by a left parenthesis, a semicolon, or one of the reserved words END, UNTIL, and ELSE is interpreted as a call to a routine with no parameters. This message is issued if the identifier was not declared as a procedure or function identifier. Note that in the current version, functions can be called with the procedure call statement.

NOTASYNCH, "routine name" is not ASYNCHRONOUS

**Information.** This message can appear as additional information on other error messages.

NOTATYPE, "symbol name" is not a type identifier

**Error.** An identifier that does not represent a type was used in a context where the compiler expected a type identifier.

NOTAVAR, "symbol name" is not declared as a variable

Error. You cannot attempt to assign a value to any object other than a variable.

NOTAVARFNID, "symbol name" is not declared as a variable or a function identifier

**Error.** You cannot attempt to assign a value to any object other than a variable or a function identifier.

NOTBEADDR, May not be parameter to ADDRESS

NOTBEASSIGN, May not be assigned

NOTBECALL, May not be called as a FUNCTION

NOTBECAST, May not be type cast

NOTBEDEREF, May not be dereferenced

NOTBEDES, May not be passed by untyped %DESCR

NOTBEEVAL, May not be evaluated

NOTBEFILOP, May not be used in a file operation

NOTBEFLD, May not be field selected

NOTBEFNCPRM, May not be passed as a FUNCTION parameter

NOTBEFORCTL, May not be used as FOR loop variable

NOTBEFORDES, May not be passed as a descriptor foreign parameter

NOTBEFOREF, May not be passed as a reference foreign parameter

NOTBEIADDR, May not be parameter to IADDRESS

NOTBEIDX, May not be indexed

NOTBEIMMED, May not be passed by untyped immediate passing mechanism

NOTBENEW, May not be written into be NEW

NOTBENSTCTL, May not be control variable for an inner FOR loop

NOTBEREAD, May not be written into be READ

NOTBEREF, May not be passed by untyped reference passing mechanism

NOTBERODES, May not be passed as a READONLY descriptor foreign parameter

NOTBEROFOR, May not be passed as a READONLY reference foreign parameter

NOTBEROVAR, May not be passed as a READONLY VAR parameter NOTBETOUCH, May not be read/modified/written

NOTBEVAR, May not be passed as a VAR parameter

NOTBEWODES, May not be passed as a WRITEONLY descriptor foreign parameter

NOTBEWOFOR, May not be passed as a WRITEONLY reference foreign parameter

NOTBEWOVAR, May not be passed as a WRITEONLY VAR parameter

NOTBEWRTV, May not be parameter to WRITEV

**Information.** These messages can appear as additional information on other error messages.

NOTBYTOFF, Field "field name" is not aligned on a byte boundary **Error.** 

NOTDECLROU, "symbol name" is not declared as a "routine." NOTINITIAL, "routine name" is not INITIALIZE

**Information.** These messages can appear as additional information on other error messages.

NOTINRNG, Value does not fall within range of the tag type

**Error.** The value specified as the case label of a variant record is not a legal value of the tag field's type. This message is also issued if a case label in a call to NEW, DISPOSE, or SIZE falls outside the range of the tag type.

NOTSAMTYP, Not the same type NOTUNBOUND, "routine name" is not UNBOUND

**Information.** These messages can appear as additional information on other error messages.

NOTVARNAM, Parameter to this predeclared function must be simple variable name

**Error.** The parameter may not be indexed, dereferenced, have a field selected, or be an expression. It must be the name of the entire variable.

NOTVOLATILE, "variable name" is non-VOLATILE

**Warning.** You should not use the ADDRESS function on a non-VOLATILE variable or component or on a formal VAR parameter.

NOUNSATDECL, No unsatisfied declaration of label "label name" in "block name"

Error.

NUMFRMLPARM, Different numbers of formal parameters

**Information.** This message can appear as additional information on other error messages.

NXTACTDIFF, NEXT of actual's component differs from that of other parameters in same section

**Error.** All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size and alignment of the array's inner dimensions.

OLDDECLSYN, Obsolete "routine" parameter declaration syntax

**Information.** The declaration of a formal routine parameter uses the obsolete Version 1 syntax. You should edit your program to incorporate the current version syntax, which is mandated by the PASCAL standard.

OPNDASSCOM, Operands are not assignment compatible OPNOTINT, Operand(s) must be of type INTEGER or UNSIGNED

Error.

ORDOPNDREQ, Ordinal operand(s) required

**Error.** This message is at warning level if you attempt to use INT, ORD, or UINT on a pointer expression. It is at error level if you use PRED or SUCC on an expression whose type is not ordinal.

OUTNOTDECL, OUTPUT not declared in heading

**Error.** A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

OVRDIVZERO, Overflow or division by zero in compile-time expression

Error.

PACKSTRUCT, "component name" of a PACKED structured type

Warning. You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writeable VAR, pass by reference, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

PARMACTDEF, Formal parameter "parameter name" has neither actual nor default

**Error.** If a formal parameter is not declared with a default, you must pass an actual parameter to it when calling its routine.

PARMCLAMAT, Parameter section classes do not match

**Information.** This message can appear as additional information on other error messages.

PARMLIMIT, VAX architectural limit of 255 parameters exceeded

**Error.** You may not declare a procedure with more than 255 formal parameters. A function whose result type requires that the result be stored in more than 64 bits or whose result type is a character string cannot have more than 254 formal parameters. In a call to a routine declared with the LIST attribute, you also cannot attempt to pass more than 255 (or 254) actual parameters.

PARMSECTMAT, Division into parameter sections does not match

**Information.** This message can appear as additional information on other error messages.

PASPREILL, Passing predeclared "routine name" is illegal

Error. You cannot use the IADDRESS function on a predeclared routine for which there is no corresponding routine in the VAX Run-Time Library (such as the interlocked functions). In addition, you cannot pass a predeclared routine as a parameter if there is no way to write the predeclared routine's formal parameter list in PASCAL. Examples of the latter case are the PRED and SUCC functions and many of the I/O routines.

PASSEXTERN, Passing mechanism allowed only on external routines **Error.** 

PCKUNPCKCON, PACKED/unpacked conflict

**Information.** This message can appear as additional information on other error messages.

POSAFTNONPOS, Positional parameter cannot follow a nonpositional parameter

Error.

POSALIGNCON, Position / alignment conflict

**Error.** The bit position specified by the POS attribute does not have the number of low-order bits implied by the specified alignment attribute.

POSINT, POS expression must be a positive INTEGER value **Error.** 

PREREQPRMLST, Passing predeclared "routine name" requires formal to include parameter list

**Error.** To pass one of the predeclared routines EXPO, ROUND, TRUNC, UNDEFINED, UTRUNC, UROUND, DBLE, SNGL, QUAD, INT, ORD, and UINT as an actual parameter to a routine, you must specify a formal parameter list in the corresponding formal routine parameter.

PRMKWNSIZ, Parameter must have a size known at compile-time

**Error.** The BIN, HEX, OCT, DEC, and UDEC functions cannot be used on conformant parameters. The SIZE and NEXT functions cannot be used on conformant parameters in compile-time constant expressions.

PROPRMEXT, Declaration of "program parameter name" is EXTERNAL—program parameter files must be locally allocated PROPRMFIL, A program parameter must be a variable of type FILE PROPRMINH, Declaration of "program parameter name" is inherited—program parameter files must be locally allocated

PROPRMLEV, Program parameter "program parameter name" is not declared as a variable at the outermost level

Error. Any external file variable (other than INPUT and OUTPUT) that is listed in the program heading must also be declared as a file variable in a VAR section in the program block.

PSECTMAXINT, Allocation of "symbol name" causes PSECT "PSECT name" to exceed MAXINT bits

**Error.** The VAX implementation restricts the size of a program section to 2,147,483,647 bits.

PTRCMPEQL, Pointer values may only be compared for equality

**Error.** The equality and inequality operators (= and < > ) are the only operators allowed for values of a pointer type; all other operators are illegal.

PTREXPRCOM, Pointer expressions are not compatible

Error. The base types of two pointer expressions being compared for equality (=) or inequality (<) are not structurally compatible.

QUOBEFEOL, Quoted string not terminated before end of line Error.

QUOSTRING, Quoted string expected

text files.

Error. The compiler expects the %DICTIONARY and %INCLUDE directives, and the radix notations for binary (%B). hexadecimal (%X), and octal constants (%O), to be followed by a quoted string of characters.

RADIXTEXT, Radix input requires FILE\_VARIABLE of type TEXT Error. The READLN and WRITELN procedures operate only on

READONLY, "variable name" is READONLY

Warning. You cannot use a READONLY variable in any context that would store a new value in the variable. For example, a READONLY variable cannot be used in a file operation.

REALCNSTRNG, Real constant out of range

**Error.** See *Programming in VAX PASCAL* for details on the range of real numbers.

REALOPNDREQ, Real (SINGLE, DOUBLE or QUADRUPLE) operand(s) required

Error.

RECHASFILE, Record contains one or more FILE components, POS is illegal

Error.

RECLENINT, RECORD\_LENGTH expression must be of type INTEGER

**Error.** The value of the record-length parameter to the OPEN procedure must be an integer.

RECLENMNGLS, RECORD\_LENGTH parameter is meaningless given file's type

**Warning.** The record-length parameter is usually relevant only for files of type TEXT and VARYING OF CHAR.

REDECL, A declaration of "symbol name" already exists in "block name"

**Error.** You cannot redeclare an identifier or a label in the same block in which it was declared. Inheriting an environment is equivalent to including all of its declarations at program or module level.

REDECLATTR, "attribute name" already specified

**Error.** Only one member of a particular attribute class can appear in the same attribute list.

REDECLFLD, Record already contains a field "field name"

**Error.** The names of the fields in a record must be unique; they cannot be duplicated between variants.

REINITVAR, "variable name" has already been initialized

**Error.** Variables are the only data items that can be initialized, and they can be initialized only once.

REPCASLAB, Value has already appeared as a label in this CASE statement

**Error.** You cannot specify the same value more than once as a case label in a CASE statement.

REPFACZERO, Repetition factor can not be the function ZERO REQCLAORNCA, Arrays and conformants of this parameter type require either CLASS\_A or CLASS\_NCA

REQCLS, Scalars and strings of this parameter type require CLASS\_S REQPKDARR, The combination of CLASS\_S and %STDESCR requires a PACKED ARRAY OF CHAR structure

REQREADVAR, READ or READV requires at least one variable to read into

REQWRITELEM, WRITE requires at least one write-list-element

**Error.** The READ and READV procedures require that you specify at least one variable to be read from a file. The WRITE procedure requires that you specify at least one item to be written to a file.

REVRNTLAB, Value has already appeared as a label in this variant part

**Error.** You cannot specify the same value more than once as a case label in a variant part of a record.

RTNSTDESCR, Routines cannot be passed using %STDESCR Error.

SENDSPR, Internal Compiler Error

**Fatal.** An error has occurred in the execution of the VAX PASCAL compiler. Along with this message, you will receive information that helps you find the location in the source program and the name of the compilation phase at which the error occurred. You may be able to rewrite the section of your program that caused the error and thus successfully compile the program. However, even if you are able to remedy the problem, please submit a Software Performance Report (SPR) to DIGITAL and provide a machine-readable copy of the program.

SEQ11FORT, PDP-11 specific directive SEQ11 treated as equivalent to FORTRAN directive

Information.

SETBASCOM, SET base types are not compatible

**Error.** The base type of two sets used in a set operation are not compatible.

SETELEORD, SET element expression must be of an ordinal type

**Error.** The expressions used to denote the elements of a set constructor or the bounds of a set type definition must have an ordinal type.

SETNOTRNG, SET element is not in range 0..255

**Error.** In a set whose base type is a subrange of integers or unsigned integers, all set elements in the set's type definition or in a constructor for the set must be in the range 0..255.

SIZACTDIFF, SIZE of actual differs from that of other parameters in same section

**Error.** All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size of the array's outermost dimension.

SIZARRNCA, Explicit size on ARRAY dimension makes CLASS\_NCA mandatory

Error.

SIZATRTYPCON, Size attribute / type conflict

Error. For an ordinal type, the size specified must be at least as large as the packed size but no larger than 32 bits. Pointer types and type SINGLE must be allocated exactly 32 bits, type DOUBLE exactly 64 bits, and type QUADRUPLE exactly 128 bits. For types ARRAY, RECORD, SET, and VARYING OF CHAR, the size specified must be at least as large as their packed sizes. For the details of allocation sizes in VAX PASCAL, see Chapter 1, VAX PASCAL System Environment.

SIZCASTYP, Variable's size conflicts with cast's target type

**Error.** In a type cast operation, the size of the variable and the size of the type to which it is cast must be identical.

SIZEDIFF, Sizes are different

**Information.** This message can appear as additional information on other error messages.

SIZEINT, Size expression must be a positive INTEGER value **Error.** 

SIZGTRMAX, Size exceeds MAXINT bits

**Error.** The size of a record or an array type or the size specified by a size attribute exceeds 3,147,483,647 bits. The VAX implementation imposes this size restriction.

SIZMULTBYT, Size of component of array passed by descriptor is not a multiple of bytes

**Error.** When an array or a conformant parameter is passed using the %DESCR mechanism specifier, the descriptor built by the compiler must follow the VAX Procedure Calling Standard. Such a descriptor can describe only an array whose components fall on byte boundaries.

SPEOVRDECL, Foreign mechanism specifier required to override parameter declaration

**Error.** When you specify a default value for a formal VAR or routine parameter, you must also use a mechanism specifier to override the characteristics of the parameter section.

SPURIOUS, "error message" at "line number"—"column number"

**Information.** The compiler did not correctly note the location of this error in your program and later could not position and print the correct error message. You may be able to correct the section of your program that caused the error and thus avoid this error. Please submit a Software Performance Report (SPR) and provide a machine-readable copy of the program if you receive this error.

SRCERRORS, Source errors inhibit continued compilation—correct and recompile

**Fatal.** A serious error previously detected in the source program has corrupted the compiler's symbol tables and inhibits further compilation. You must correct the serious error and recompile the program.

SRCTXTIGNRD, Source text following end of compilation unit ignored

**Warning.** The compiler ignores any text following the "END." that terminates a compilation unit. This error probably resulted from an unmatched END in your program.

STDACTINCMP, Nonstandard: actual is not name compatible with other parameters in same section

**Information.** According to the Pascal standard, all actual parameters passed to a parameter section must have the same type identifier or the same type definition. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDATTRLST, Nonstandard: attribute list

STDBIGLABEL, Nonstandard: label number greater than 9999

STDBLANKPAD, Nonstandard: blank-padding used during string operation

STDCALLFUNC, Nonstandard: function "function name" called as a procedure

STDCAST, Nonstandard: type cast operator

STDCNFARR, Nonstandard: conformant array syntax

STDCONCAT, Nonstandard: concatenation operator

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCONST, Nonstandard: "type name" constant

**Information.** Binary, hexadecimal, and octal constants and constants of type DOUBLE, QUADRUPLE, and UNSIGNED are VAX extensions to PASCAL. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCTLDECL, Nonstandard: control variable "variable name" not declared in VAR section of "block name"

Information. The Pascal standard requires that the control variable of a FOR statement be declared in the same block in which the FOR statement appears.

STDDECLSEC, Nonstandard: declaration sections either out of order or duplicated in "block name"

**Information.** In the Pascal standard, the declaration sections must appear in the order LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION. The ability to specify the sections in any order is a VAX extension. This message occurs only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDEFPARM, Nonstandard: default parameter declaration

**Information.** This message refers to VAX extensions to PASCAL and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDIRECT, Nonstandard: "directive name" directive

Information. The EXTERN, EXTERNAL, FORTRAN, and SEQ11 directives are VAX extensions to PASCAL. (FORWARD is the only directive specified by the Pascal standard.) This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPCASLST, Nonstandard: empty case-list element

Information. This message is issued if you do not specify any case labels and executable statements between two semicolons or between OF and a semicolon in the CASE statement. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDEMPPARM, Nonstandard: empty actual parameter position

Information. This message refers to VAX extensions to PASCAL and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPREC, Nonstandard: empty record section

**Information.** The Pascal standard does not allow record type definitions of the form RECORD END;. This message appears only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPSTR, Nonstandard: empty string

**Information.** This message refers to VAX extensions to PASCAL and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPVRNT, Nonstandard: empty variant

**Information.** This message occurs if you do not specify a variant between two semicolons or between OF and a semicolon. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDEXPON, Nonstandard: exponentiation operator STDEXTSTR, Nonstandard: extended string syntax STDFORMECH, Nonstandard: foreign mechanism specifier

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDFUNCTRES, Nonstandard: FUNCTION returning a value of a "type name" type

**Information.** The ability of functions to have structured result types is a VAX extension to PASCAL. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDINCLUDE, Nonstandard: %INCLUDE directive STDINITVAR, Nonstandard: initialization syntax in VAR section

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDMATCHVRNT, Nonstandard: no matching variant label

**Information.** This message is issued if you call the NEW or DISPOSE procedure, and one of the case labels specified in the call does not correspond to a case label in the record variable. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDMODCTL, Nonstandard: potential uplevel modification of "variable name" prohibits use as control variable

**Information.** You cannot use as the control variable of a FOR statement any variable that might be modified in a nested block. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDMODULE, Nonstandard: MODULE declaration

**Information.** The item listed in this message is a VAX extension to PASCAL. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDNILCON, Nonstandard: use of reserved word NIL as a constant

**Information.** Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, and constants of BOOLEAN and enumerated types. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDNOFRML, Nonstandard: FUNCTION or PROCEDURE parameter declaration lacks formal parameter list

**Information.** This message is issued if you try to pass actual parameters to a formal routine parameter for which you declared no formal parameter list. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDNONPOS, Nonstandard: nonpositional parameter syntax STDOTHER, Nonstandard: OTHERWISE clause STDPASSPRE, Nonstandard: passing predeclared "routine name"

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPCKSET, Nonstandard: combination of PACKED and unpacked sets

**Information.** The Pascal standard does not allow PACKED and unpacked sets to be combined in set operations. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPREDECL, Nonstandard: predeclared "routine"

**Information.** Many predeclared procedures and functions are VAX extensions to PASCAL. The use of these routines causes this message to be issued if you have specified the /STANDARD qualifier with the PASCAL command.

STDPRETYP, Nonstandard: predefined type "type name"

**Information.** The types SINGLE, DOUBLE, QUADRUPLE, UNSIGNED, and VARYING OF CHAR are VAX extensions to PASCAL. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDRADIX, Nonstandard: radix constant

**Information.** The item listed in this message is a VAX extension to PASCAL. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDRDBIN, Nonstandard: binary input from a TEXT file STDRDENUM, Nonstandard: enumerated type input from a TEXT file STDRDHEX, Nonstandard: hexadecimal input from a TEXT file STDRDOCT, Nonstandard: octal input from a TEXT file STDRDSTR, Nonstandard: string input from a TEXT file

**Information.** The Pascal standard for PASCAL allows only INTEGER, CHAR, and REAL values to be read from a text file. The ability to read values of other types is a VAX extension to PASCAL. These messages are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDREDECLNIL, Nonstandard: redeclaration of reserved word NIL

**Information.** The Pascal standard considers NIL a reserved word, while VAX PASCAL considers it to be a predeclared identifier. Thus, if you have specified the /STANDARD qualifier with the PASCAL command, this message will be issued if you attempt to redefine NIL.

STDREM, Nonstandard: REM operator

**Information.** The item listed in this message is a VAX extension to PASCAL. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSIMCON, Nonstandard: only simple constant (optional sign) or quoted string

**Information.** Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, and constants of BOOLEAN and enumerated types. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSPECHAR, Nonstandard: "\$" or "\_" in identifier STDSTRCOMPAT, Nonstandard: string compatibility

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSTRUCT, Nonstandard: types do not have same name

**Information.** Because the Pascal standard does not recognize structural compatibility, two types must have the same type identifier or type definition to be compatible. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSYMLABEL, Nonstandard: symbolic label

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDTAGFLD, Nonstandard: invalid use of tag field

**Information.** The tag field of a variant record cannot be a parameter to the ADDRESS function, nor can you pass it as a writeable VAR, %REF, %DESCR, or %STDESCR formal parameter. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDUNSAFE, Nonstandard: UNSAFE compatibility

**Information.** If you have used the UNSAFE attribute on an object that is later tested for compatibility, you will receive this message. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDUSEDCNF, Nonstandard: conformant array used as a string STDUSEDPCK, Nonstandard: PACKED ARRAY [1..1] OF CHAR used as a string

STDVALUE, Nonstandard: VALUE initialization section STDVAXCDD, Nonstandard: %DICTIONARY directive

**Information.** These messages refer to VAX extensions to PASCAL and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVRNTPART, Nonstandard: empty variant part

**Information.** According to the Pascal standard, a variant part that declares no case labels and field lists between the words OF and END is illegal. This message occurs only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVRNTRNG, Nonstandard: variant labels do not cover the range of the tag type

**Information.** According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDWRTBIN, Nonstandard: binary output to a TEXT file STDWRTENUM, Nonstandard: user defined enumerated type output to a TEXT file

STDWRTHEX, Nonstandard: hexadecimal output to a TEXT file STDWRTOCT, Nonstandard: octal output to a TEXT file

Information. The Pascal standard allows only INTEGER, BOOLEAN, CHAR, REAL, and PACKED ARRAY [1..n] OF CHAR values to be written to a text file. The ability to write values of other types is a VAX extension to PASCAL. These messages are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STREQLLEN, String values must be of equal length

Error. You cannot perform string comparisons on character strings that have different lengths.

STROPNDREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) operand required

STRPARMREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) parameter required

STRTYPREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) type required

Error. The file-name parameter to the OPEN procedure and the parameter to the LENGTH function must be character strings of the types listed.

SYNASCII, Illegal ASCII character

SYNASSERP, Syntax: ":=", ";" or ")" expected

SYNASSIGN, Syntax: ":=" expected

SYNASSSEMI, Syntax: ":=" or ";" expected

SYNATRCAST, Syntax: attribute list not allowed on a type cast

SYNATTTYPE, Syntax: attribute-list or type specification SYNBEGDECL, Syntax: BEGIN or declaration expected

SYNBEGIN, Syntax: BEGIN expected

SYNCOASSERP, Syntax: ",", ":=", ";" or ")" expected SYNCOELRB, Syntax: ",", "..." or "]" expected

SYNCOLCOMRP, Syntax: ":", "," or ")" expected

SYNCOLON, Syntax: ":" expected

SYNCOMCOL, Syntax: "," or ":" expected

SYNCOMDO, Syntax: "," or DO expected SYNCOMMA, Syntax: "," expected

SYNCOMRB, Syntax: "," or "]" expected SYNCOMRP, Syntax: "," or ")" expected SYNCOMSEM, Syntax: "," or ";" expected

SYNCONTMESS, Syntax: CONTINUE or MESSAGE expected

SYNCOSERP, Syntax: ",", ";" or ")" expected SYNDIRBLK, Syntax: directive or block expected

> Error. The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

## SYNDIRMIS, Syntax: directive missing, EXTERNAL assumed

**Error.** In the absence of a directive where one is expected, the compiler assumes that EXTERNAL is the intended directive and proceeds with compilation based on that assumption.

SYNDO, Syntax: DO expected SYNELIPSIS, Syntax: ".." expected

SYNELSESTMT, Syntax: ELSE or start of new statement expected

SYNEND, Syntax: END expected SYNEQL, Syntax: "=" expected

SYNERRCTE, Error in compile-time expression

SYNEXPR, Syntax: expression expected

SYNEXSEOTEN, Syntax: expression, ";", OTHERWISE or END expected

SYNFUNPRO, Syntax: FUNCTION or PROCEDURE expected

SYNHEADTYP, Syntax: routine heading or type identifier expected

SYNIDCAEND, Syntax: identifier, CASE or END expected SYNIDCARP, Syntax: identifier, CASE or ")" expected

SYNIDCASE, Syntax: identifier or CASE expected SYNIDENT, Syntax: identifier expected

SYNILLEXPR, Syntax: ill-formed expression

SYNINT, Syntax: integer expected

SYNINVSEP, Syntax: invalid token separator SYNIVATRLST, Syntax: illegal attribute list SYNIVPARM, Syntax: illegal actual parameter

SYNIVPRMLST, Syntax: illegal actual parameter list

SYNIVSYM, Syntax: illegal symbol SYNIVVAR, Syntax: illegal variable SYNLABEL, Syntax: label expected SYNLBRAC, Syntax: "[" expected SYNLPAREN, Syntax: "(" expected SYNLPAREN, Syntax: "(" expected "" expected """

SYNLPASEM, Syntax: "(" or ";" expected SYNLPCORB, Syntax: "(", "," or "]" expected SYNLPSECO, Syntax: "(", ";" or ":" expected

SYNMECHEXPR, Syntax: mechanism specifier or expression expected

SYNNEWSTMT, Syntax: start of new statement expected

SYNOF, Syntax: OF expected

SYNPARMLST, Syntax: actual parameter list

SYNPARMSEC, Syntax: parameter section expected

SYNPROMOD, Syntax: PROGRAM or MODULE expected

SYNQUOSTR, Syntax: quoted string expected

SYNRBRAC, Syntax: "]" expected

SYNRESWRD, Syntax: reserved word cannot be redefined

SYNRPAREN, Syntax: ")" expected SYNRPASEM, Syntax: ";" or ")" expected

SYNRTNTYPCNF, Syntax: routine heading, type identifier or conformant schema expected

SYNSEMI, Syntax: ";" expected

SYNSEMIEND, Syntax: ";" or END expected

SYNSEMMODI, Syntax: ";", "::", "^", or "[" expected

SYNSEMRB, Syntax: ";" or "]" expected

SYNSEOTEN, Syntax: ";", OTHERWISE or END expected

SYNTHEN, Syntax: THEN expected

SYNTODOWN, Syntax: TO or DOWNTO expected

SYNTYPCNF, Syntax: type identifier or conformant schema expected

SYNTYPID, Syntax: type identifier expected

**Error.** The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNTYPPACK, Only ARRAY, FILE, RECORD or SET types can be PACKED

**Warning.** You cannot attempt to pack any type other than the structured types listed in the message.

SYNTYPSPEC, Syntax: type specification expected

SYNUNEXDECL, Syntax: declaration encountered in executable section

SYNUNTIL, Syntax: UNTIL expected

SYNXTRASEMI, Syntax: "; ELSE" is not valid Pascal, ELSE matched with IF on line "line number"

**Error.** The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

TAGNOTORD, Tag type must be an ordinal type

**Error.** The type of a variant record's tag field must be one of the ordinal types.

TOOIDXEXPR, Too many index expressions; type has only "number of dimensions" dimensions

**Error.** A call to the UPPER or LOWER function specified an index value that exceeds the number of dimensions in the dynamic array.

TYPFILSIZ, Type contains one or more FILE components, size attribute is illegal

**Error.** The allocation size of a FILE type cannot be controlled by a size attribute; therefore, you cannot use a size attribute on any type that has a file component.

TYPHASFILE, Type contains one or more FILE components

**Error.** Many operations are illegal on objects of type FILE and objects of structured types with file components; for example, you cannot initialize them, use them as value parameters, or read them with input procedures.

TYPHASNOVRNT, Type contains no variant part

**Error.** The formats of the NEW, DISPOSE, and SIZE routines that allow case labels to be specified can be used only when their parameters have variants.

TYPPTRFIL, Type must be pointer or FILE

**Error.** You cannot use the syntax "variable" to refer to an object whose type is not pointer or FILE.

TYPSTDESCR, %STDESCR not allowed for this type

**Error.** The %STDESCR mechanism specifier is allowed only on objects of type CHAR, PACKED ARRAY [1..n] OF CHAR, VARYING OF CHAR, and arrays of these types.

TYPVARYCHR, Component type of VARYING must be CHAR Error.

UNALIGNED, "variable name" is UNALIGNED

Warning. You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writeable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

UNBPNTRET, "routine name" is not UNBOUND—only 32-bit address of entry point returned

Warning. The IADDRESS function returns an address as an integer. If you pass it the name of a routine, IADDRESS returns only the first 32 bits of the routine's bound procedure value.

UNDECLFRML, Undeclared formal parameter "symbol name"

**Error.** A formal parameter name listed in a nonpositional call to a routine does not match any of the formal parameters declared in the routine heading.

UNDECLID, Undeclared identifier "symbol name"

Error. In PASCAL, an identifier must be declared before it is used. There are no default or implied declarations.

UNSCNFVRY, UNSAFE attribute not allowed on conformant VARYING schema

Error.

UNSEXCRNG, UNSIGNED constant exceeds range

Error. The largest value allowed for an UNSIGNED integer is 4,294,967,295.

UPLEVELACC, Unbound "routine name" precludes uplevel access to "variable name"

Error. A routine that was declared with the UNBOUND attribute cannot attempt to refer to automatic variables, routines, or labels declared in outer blocks.

UPLEVELGOTO, Unbound "routine name" precludes uplevel GOTO to "label name"

**Error.** A routine that was declared with the UNBOUND attribute cannot attempt to refer to automatic variables, routines, or labels declared in outer blocks.

USEDBFDECL, "symbol name" was used before being declared Warning.

V1DYNARR, Decommitted Version 1 dynamic array type

**Warning.** The type syntax used to define a dynamic array parameter has been decommitted for the current version of VAX PASCAL. You should edit your program to make the type definition conform to the Version 3 conformant array syntax.

V1DYNARRASN, Decommitted Version 1 dynamic array assignment

**Warning.** In Version 1, dynamic arrays used in assignments could not be checked for compatibility until run-time. This warning indicates that your program depends on an obsolete feature, which you should consider changing to reflect the current version syntax for conformant array parameters.

V1MISSPARM, Decommitted missing parameter syntax: correct by adding "number of commas" comma(s)

**Warning.** An OPEN procedure called with the decommitted Version 1 syntax fails to mark omitted parameters with commas. Your program depends on this obsolete feature, and you should insert the correct number of commas as listed in the message.

V1PARMSYN, Use of unsupported V1 omitted parameter syntax with a Version 3 feature(s)

**Error.** In a parameter list for the OPEN procedure, you cannot use both the Version 1 syntax for OPEN and the parameters that are new to the current version of VAX PASCAL.

V1RADIX, Decommitted Version 1 radix output specification

**Information.** In Version 1, octal and hexadecimal values could be written by placing the keywords OCT or HEX after a field width expression. Your program uses this obsolete feature which you should consider changing to use the current versions OCT or HEX predeclared functions.

VALOUTBND, Value to be assigned is out of bounds

**Error.** A value specified in an array or record constructor exceeds the subrange defined as the type of the corresponding component.

VALUEINIT, VALUE variables must be initialized

Error. Variables with both the VALUE and GLOBAL attributes must be given an initial value in either the VAR section or in the VALUE section.

VALUETOOBIG, VALUE attribute not allowed on objects larger than 32 bits

**Error.** Variables with the VALUE attribute cannot be larger than 32 bits since they are expressed to the linker as Global Symbol References.

VALUETYP, VALUE allowed only on ordinal or real types Error.

VALUEVISIB, GLOBAL or EXTERNAL visibility is required with the VALUE attribute

Error. Variables with the VALUE attribute must be given either EXTERNAL or GLOBAL visibility. (If the variable is given GLOBAL visibility, then it must also be given an initial value.)

VARCOMFRML, Variable is not compatible with formal parameter "formal parameter name"

Error. A variable being passed as an actual parameter is not compatible with the corresponding formal parameter indicated. Variable parameters must be structurally compatible. The reason for the incompatibility is provided in an informational message that the compiler prints along with this error message.

VARNOTEXT, Variable must be of type TEXT

**Error.** The EOLN function requires that its parameter be a file of type TEXT.

VARPRMRTN, Formal VAR parameter may not be a routine

**Error.** The reserved word VAR cannot precede the word PROCEDURE or FUNCTION in a formal parameter declaration.

VARPTRTYP, Variable must be of a pointer type

**Error.** The NEW and DISPOSE procedures operate only on pointer variables.

VARYFLDS, LENGTH and BODY are the only fields in a VARYING type

**Error.** You cannot use the syntax "variable.identifier" to specify any fields of a VARYING OF CHAR variable other than LENGTH and BODY.

VISAUTOCON, Visibility / AUTOMATIC allocation conflict

**Error.** The GLOBAL, EXTERNAL, WEAK\_GLOBAL, and WEAK\_EXTERNAL attributes require static allocation and therefore conflict with the AUTOMATIC attribute.

VISGLOBEXT, Visibilities are not GLOBAL/EXTERNAL or EXTERNAL/EXTERNAL

**Information.** In repeated declarations of a variable or routine, only one declaration at most may be GLOBAL; all others must be EXTERNAL. This message can appear as additional information for other error messages.

WDTHONREAL, Second field width is allowed only when value is of a real type

**Error.** The "fraction" value in a field-width specification is allowed only for real-number values.

#### WRITEONLY, "variable name" is WRITEONLY

**Warning.** You cannot use a WRITEONLY variable in any context that requires the variable to be evaluated. For example, a WRITEONLY variable cannot be used as the control variable of a FOR statement.

XTRAERRORS, Additional diagnostics occurred on this line

**Information.** The number of errors occurring on this line exceeds implementation's limit for outputting errors. You should correct the errors given and recompile your program.

# A.2 Run-Time Diagnostics

When an error occurs at run-time, the VAX PASCAL run-time system issues an error message and aborts program execution. The syntax of the error message is as follows:

%PAS-F-code, text

#### code

An abbreviation of the message text. Messages are alphabetized in this appendix by this code.

#### text

The explanation of the error.

Some conditions, particularly I/O errors, may cause several messages to be generated. The first message is a diagnostic that specifies the file that was being accessed (if any) when the error occurred and the nature of the error. Next, a VAX RMS error message may be generated. In most cases, you should be able to understand the error by looking up the first message in the following list. If not, refer to the VAX/VMS System Messages and Recovery Procedures Reference Manual for an explanation of the VAX RMS error message.

Quotation marks ("") in message text enclose items for which the compiler will replace with the name of a data object when it generates the message.

ACCMETINC, ACCESS\_METHOD specified is incompatible with this file

**Explanation.** The value of the ACCESS\_METHOD parameter for a call to the OPEN procedure is not compatible with the file's organization or record type. You can use DIRECT access only with files that have relative organization or sequential organization and fixed-length records. You can use KEYED access only with indexed files.

**User Action.** Make sure that you are accessing the correct file. Consult Chapter 4, Input and Output with RMS of this manual to determine which access method you should use.

AMBVALENU, "string" is an ambiguous value for enumerated type "type"

**Explanation.** While a value of an enumerated type was being read from a text file, not enough characters of the identifier were found to specify an unambiguous value.

**User Action.** Specify enough characters of the identifier so that it is not ambiguous.

ARRINDVAL, array index value is out of range

**Explanation.** You enabled BOUNDS checking for a compilation unit and attempted to specify an index that is outside the array's index bounds.

**User Action.** Correct the program or data so that all references to array indexes are within the declared bounds.

ARRNOTCOM, conformant array is not compatible

**Explanation.** You attempted to assign one dynamic array to another that did not have the same index bounds. This error occurs only when the arrays use the decommitted Version 1 syntax for dynamic array parameters.

**User Action.** Correct the program so that the two dynamic arrays have the same index bounds. You could also change the arrays to conform to the current syntax for conformant arrays; most incompatibilities could then be detected at compile time rather than at run-time. See *Programming in VAX PASCAL* for more information on current conformant arrays.

ARRNOTSTR, conformant array is not a string

**Explanation.** In a string operation, you used a conformant PACKED ARRAY OF CHAR whose index had a lower bound not equal to 1 or an upper bound greater than 65535.

**User Action.** Correct the array's index so that the array is in fact a character string.

BUGCHECK, internal consistency failure "nnn" in Pascal Run-Time Library

**Explanation.** The Pascal Run-Time Library has detected an internal error or inconsistency. This problem may be caused by an out-of-bounds array reference or a similar error in your program.

User Action. Rerun your program with all CHECK options enabled. If you are unable to find an error in your program, please submit a Software Performance Report (SPR) to DIGITAL, including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

BADBITARG, Nbits argument to DEC or UDEC must be between 1 and 32

**Explanation.** The value being passed to DEC or UDEC exceeds the boundaries of a longword (32 bits).

**User Action.** If you are calling PAS\$DEC or PAS\$UDEC directly, examine your program for bad data. If PAS\$DEC or PAS\$UDEC have been called through predeclared routines in your program, then submit an SPR.

CANCNTERR, handler cannot continue from a nonfile error

**Explanation**. A user condition handler attempted to return SS\$\_ CONTINUE for a nonfile error. The only allowed user recovery action for a nonfile error is (1) an uplevel GOTO unwinding.

**User Action.** Modify the user handler to use either one of the allowed recovery actions for nonfile errors, or to resignal the error if no recovery action is possible.

CASSELVAL, CASE selector value is out of range

**Explanation.** The value of the case selector in a CASE statement does not equal any of the specified case labels, and the statement has no OTHERWISE clause.

**User Action.** Either add an OTHERWISE clause to the CASE statement or change the value of the case selector so that it equals one of the case labels. See *Programming in VAX PASCAL* for more information.

CONCATLEN, string concatenation has more than 65535 characters

**Explanation.** The result of a string concatenation operation would result in a string longer than 65,535 characters, which is the maximum length of a string.

**User Action.** Correct the program so that all concatenations result in strings no longer than 65,535 characters.

CURCOMUND, current component is undefined for DELETE or UPDATE

**Explanation.** You attempted a DELETE or UPDATE procedure when no current component was defined. A current component is defined by a successful GET, FIND, FINDK, RESET, or RESETK that locks the component. Files opened with HISTORY:=READONLY never lock components.

**User Action.** Correct the program so that a current component is defined before executing DELETE or UPDATE.

DELNOTALL, DELETE is not allowed for a sequential organization file

**Explanation.** You attempted a DELETE procedure for a file with sequential organization, which is not allowed. DELETE is valid only on files with relative or indexed organization.

**User Action.** Make sure that the program is referencing the correct file. Consult Chapter 4, Input and Output with RMS, to determine what file characteristics are appropriate for your application.

#### ERRDURCLO, error during CLOSE

Explanation. VAX RMS reported an unexpected error during execution of the CLOSE procedure. The RMS error message is also displayed. This message may also be issued with Error severity when files are implicitly closed during a procedure or image exit.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for the description of the VAX RMS error.

## ERRDURDEL, error during DELETE

**Explanation.** VAX RMS reported an unexpected error during execution of a DELETE procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for the description of the VAX RMS error.

## ERRDURDIS, error during DISPOSE

**Explanation**. An error occurred during execution of a DISPOSE procedure. An additional message that further describes the error may also be displayed.

**User Action.** Make sure that the heap storage being freed was allocated by a successful call to the NEW procedure, and that it has not been already freed. If an additional message is shown, see the VAX/VMS System Messages and Recovery Procedures Reference Manual for the description of that message.

## ERRDUREXT, error during EXTEND

Explanation. VAX RMS reported an unexpected error during execution of an EXTEND procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for the description of the VAX RMS error.

## ERRDURFIN, error during FIND or FINDK

**Explanation.** VAX RMS reported an unexpected error during execution of a FIND or FINDK procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for the description of the VAX RMS error.

## ERRDURGET, error during GET

**Explanation.** VAX RMS reported an unexpected error during execution of the GET procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for the description of the RMS error.

## ERRDURMAR, error during MARK

**Explanation.** An error occurred during execution of the PAS\$MARK2 procedure. An additional message is displayed that further describes the error.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the additional message.

# ERRDURNEW, error during NEW

**Explanation.** An error occurred during execution of the NEW procedure. An additional message is displayed that further describes the error.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the additional message.

## ERRDUROPE, error during OPEN

**Explanation.** An unexpected error occurred during execution of the OPEN procedure, or during an implicit open caused by a RESET or REWRITE procedure. An additional message is displayed that further describes the error.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the additional message.

## ERRDURPRO, error during prompting

**Explanation.** VAX RMS reported an unexpected error during output of partial lines to a terminal. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

## ERRDURPUT, error during PUT

**Explanation.** VAX RMS reported an unexpected error during execution of the PUT procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS message.

# ERRDURREL, error during RELEASE

**Explanation.** An unexpected error occurred during execution of the PAS\$RELEASE2 procedure. An additional message may be displayed that further describes the error.

**User Action.** Make sure that the marker argument was returned from a successful call to PAS\$MARK2 and that the storage has not been already freed. If an additional message is displayed, see the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of that message.

## ERRDURRES, error during RESET or RESETK

**Explanation.** VAX RMS reported an unexpected error during execution of the RESET or RESETK procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

## ERRDURREW, error during REWRITE

**Explanation.** VAX RMS reported an unexpected error during execution of the REWRITE procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

## ERRDURTRU, error during TRUNCATE

**Explanation.** VAX RMS reported an unexpected error during execution of the TRUNCATE procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

## ERRDURUNL, error during UNLOCK

**Explanation.** VAX RMS reported an unexpected error during execution of the UNLOCK procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

# ERRDURUPD, error during UPDATE

**Explanation.** VAX RMS reported an unexpected error during execution of the UPDATE procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

#### ERRDURWRI, error during WRITELN

**Explanation.** VAX RMS reported an unexpected error during execution of the WRITELN procedure. The RMS error message is also displayed.

**User Action.** See the VAX/VMS System Messages and Recovery Procedures Reference Manual for a description of the RMS error.

#### EXTNOTALL, EXTEND is not allowed for a shared file

**Explanation.** You attempted an EXTEND procedure for a file for which the program did not have exclusive access. EXTEND requires that no other users be allowed to access the file. Note that this message may also be issued if you do not have permission to extend to the file.

**User Action.** Correct the program so that the file is opened with SHARING:=NONE, which is the default, before performing an EXTEND.

## FAIGETLOC, failed to GET locked component

**Explanation.** You attempted to access a component of a file that was locked by another user. You can usually expect this condition to occur when more than one user is accessing the same relative or indexed file.

**User Action.** Determine whether this condition should be allowed to occur. If so, modify your program so that it detects the condition and retries the operation later. See Chapter 4, Input and Output with RMS, for more information.

## FILALRACT, file "file name" is already active

**Explanation.** You attempted a file operation on a file for which another operation was still in progress. This error can occur if a file is used in AST or condition-handling routines.

**User Action.** Modify your program so that it does not attempt to use files that may currently be in use.

#### FILALRCLO, file is already closed

**Explanation.** You attempted to close a file that was already closed.

**User Action.** Modify your program so that it does not attempt to close files that are not open.

## FILALROPE, file is already open

**Explanation.** You attempted to open a file that was already open.

**User Action.** Modify your program so that it does not attempt to open files that are already open.

# FILNAMREQ, FILE\_NAME required for this HISTORY or DISPOSITION

**Explanation.** You attempted to open a nonexternal file without specifying a FILE\_NAME parameter to the OPEN procedure, but the HISTORY or DISPOSITION parameter specified requires a file name. If HISTORY is OLD or READONLY, or if DISPOSITION is PRINT, PRINT\_DELETE, SUBMIT, or SUBMIT\_DELETE, you must also specify FILE\_NAME.

**User Action.** Add a FILE\_NAME parameter to the OPEN procedure call, specifying an appropriate file name.

# FILNOTDIR, file is not opened for direct access

**Explanation.** You attempted to execute a DELETE, FIND, LOCATE, or UPDATE procedure on a file that was not opened for direct access.

**User Action.** Modify the program to specify the ACCESS\_METHOD:=DIRECT parameter to the OPEN procedure when opening the file. Consult Chapter 4, Input and Output with RMS, to see if direct access is appropriate for your application.

#### FILNOTFOU, file not found

**Explanation.** You attempted to open a file that does not exist. An additional VAX RMS message is displayed that further describes the problem.

**User Action.** Make sure that you are specifying the correct file. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

#### FILNOTGEN, file is not in Generation mode

**Explanation.** You attempted a file operation that required the file to be in Generation mode (ready for writing).

**User Action.** Modify the program to use a REWRITE, TRUNCATE, or LOCATE procedure to place the file in Generation mode as appropriate. See Chapter 4, Input and Output with RMS, Input and Output with RMS, for more information.

## FILNOTINS, file is not in Inspection mode

**Explanation.** You attempted a file operation that required the file to be in Inspection mode (ready for reading).

**User Action.** Modify the program to use a RESET, RESETK, FIND, or FINDK procedure to place the file in Inspection mode as appropriate. See Chapter 4, Input and Output with RMS, for more information.

# FILNOTKEY, file is not opened for keyed access

**Explanation.** You attempted to execute a FINDK, RESETK, DELETE, or UPDATE procedure on a file that was not opened for keyed access.

**User Action.** Modify the program to specify the ACCESS\_METHOD:=KEYED parameter to the OPEN procedure when opening the file. Consult Chapter 4, Input and Output with RMS, to make sure that keyed access is appropriate to your application.

#### FILNOTOPE, file is not open

**Explanation.** You attempted to execute a file manipulation procedure on a file that was not open.

**User Action.** Correct the program to open the file using a RESET, REWRITE, or OPEN procedure as appropriate. See Chapter 4, Input and Output with RMS, for more information.

## FILNOTSEQ, file is not sequential organization

**Explanation.** You attempted to execute the TRUNCATE procedure on a file that does not have sequential organization. TRUNCATE is valid only on sequential files.

**User Action.** Make sure that your program is accessing the correct file. Correct the program so that all TRUNCATE operations are performed on sequential files.

#### FILNOTTEX, file is not a textfile

**Explanation.** You performed a file operation that required a file of type TEXT on a nontext file. Note that the type FILE OF CHAR is not equivalent to TEXT unless you have compiled the program with the /OLD\_VERSION qualifier.

**User Action.** Make sure that your program is accessing the correct file. Correct the program so that a text file is always used when required.

GENNOTALL, Generation mode is not allowed for a READONLY file

**Explanation.** You attempted to place a file declared with the READONLY attribute into Generation mode, which is not allowed. Note that the READONLY file attribute is not equivalent to the HISTORY:=READONLY parameter to the OPEN procedure.

**User Action.** Correct the program so that the file either does not have the READONLY attribute or is not placed into Generation mode.

## GETAFTEOF, GET attempted after end-of-file

**Explanation.** You attempted a GET operation on a file while EOF(f) was TRUE. This situation occurred when a previous GET operation (possibly implicitly performed by a RESET, RESETK, or READ procedure) read to the end of the file and caused the EOF(f) function to return TRUE. If another GET is then performed, this error is given.

**User Action.** Correct the program so that it either tests for EOF(f) being TRUE before attempting a GET operation or repositions the file before the end-of-file marker.

## GOTOFAILED, non-local GOTO failed

**Explanation.** An error occurred while a nonlocal GOTO was being performed. This error might occur because of an error in the user program, such as an out-of-bounds array reference.

**User Action.** Rerun your program, enabling all CHECK options. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to DIGITAL, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

## HALT, HALT procedure called

**Explanation.** The program terminated its execution by executing the HALT procedure. This message is solely informational.

User Action. None.

INSNOTALL, Inspection mode is not allowed for a WRITEONLY file

**Explanation.** You attempted to place a file declared with the WRITEONLY attribute into Inspection mode, which is not allowed.

**User Action.** Correct the program so that the file variable either does not have the WRITEONLY attribute or is not placed into Inspection mode.

### INSVIRMEM, insufficient virtual memory

**Explanation.** The VAX Run-Time Library was unable to allocate enough heap storage to open the file.

**User Action.** Examine your program to see whether it is making excessive use of heap storage, which might be allocated using the NEW procedure or the Run-Time Library procedure LIB\$GET\_VM. Modify your program to free any heap storage it does not need.

# INVARGPAS, invalid argument to Pascal Run-Time Library

**Explanation.** An invalid argument or inconsistent data structure was passed to the VAX Run-Time Library by the compiled code, or a system service returned an unrecognized value to the Run-Time Library.

**User Action.** Rerun your program with all CHECK options enabled. Make sure that the version of the current operating system is compatible with the version of the compiler. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to DIGITAL, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

## INVFILSYN, invalid file name syntax

**Explanation.** You attempted to open a file with an invalid file name. The file name used can be derived from the file variable name, the value of the FILE\_NAME parameter to the OPEN procedure, or the logical name translations (if any) of the file variable name and portions of the FILE\_NAME parameter and your default device and directory. The displayed text may include the erroneous file name. This error can also occur if the value of the FILE\_NAME parameter is longer than 255 characters. Additional VAX RMS messages may be displayed that further describe the error.

**User Action.** Use the information provided in the displayed message(s) to determine which component of the file name is invalid. Verify that any logical names used are defined correctly. See *Programming in VAX PASCAL* for information on file names.

## INVFILVAR, invalid file variable at location "nnn"

**Explanation.** The file variable passed to a VAX Run-Time Library procedure was invalid or corrupted. This problem might occur because of an error in the user program, such as an out-of-bounds array access. It can also occur if a file variable is passed from a routine compiled with a version of VAX PASCAL earlier than Version 2 to a routine compiled with a later version of the compiler.

**User Action.** Rerun your program with all CHECK options enabled, and recompile all modules using the same compiler. If the problem persists, please submit a Software Performance Report (SPR) to DIGITAL, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

# INVKEYDEF, invalid key definition

**Explanation.** You attempted to open a file of type RECORD whose component type contained a field with an invalid KEY attribute. One of the following errors occurred:

- A new file was being created and the key numbers were not dense.
- A key field was defined at an offset of more than 65,535 bytes from the beginning of the record.

User Action. If a new file is being created, make sure that the key fields are numbered consecutively, starting with 0 for the required primary key. If you are opening an existing file, you must explicitly specify HISTORY:=OLD or HISTORY:=READONLY as a parameter to the OPEN procedure. Make sure that the length of the record is within the maximum permitted for the file organization being used. Consult Chapter 4, Input and Output with RMS, for more information.

INVRECLEN, invalid record length of "nnn"

**Explanation.** A file was being opened, and one of the following errors occurred:

- The length of the file components was greater than 65,535 bytes.
- The value of the RECORD\_LENGTH parameter to the OPEN procedure was greater than 65,535.

**User Action.** Correct the program so that the record length used is within the permitted limits for the type of file being used. Consult Chapter 4, Input and Output with RMS, for more information.

INVSYNBIN, "string" is invalid syntax for a binary value

**Explanation.** While a READ or READV procedure was reading a binary value from a text file, the characters read did not conform to the syntax for a binary value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

INVSYNHEX, "string" is invalid syntax for a hexadecimal value

**Explanation.** While a READ or READV procedure was reading a hexadecimal value from a text file, the characters read did not conform to the syntax for an hexadecimal value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

INVSYNENU, "string" is invalid syntax for an enumerated value

**Explanation.** While a READ or READV procedure was reading an identifier of an enumerated type from a text file, the characters read did not conform to the syntax for an enumerated value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

INVSYNINT, "string" is invalid syntax for an integer value

**Explanation.** While a READ or READV procedure was reading a value for an integer identifier from a text file, the characters read did not conform to the syntax for an integer value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

INVSYNOCT, "string" is invalid syntax for an octal value

**Explanation.** While a READ or READV procedure was reading an octal value from a text file, the characters read did not conform to the syntax for an octal value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

INVSYNREA, "string" is invalid syntax for a real value

**Explanation.** While a READ or READV procedure was reading a value for a real identifier from a text file, the characters read did not conform to the syntax for a real value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

INVSYNUNS, "string" is invalid syntax for an unsigned value

**Explanation.** While a READ or READV procedure was reading a value for an unsigned identifier from a text file, the characters read did not conform to the syntax for an unsigned value. The displayed message includes the text actually read and the record number in which this text occurred.

**User Action.** Correct the program or the input data so that the correct syntax is used. Consult *Programming in VAX PASCAL* for more information.

KEYCHANOT, key field change is not allowed

**Explanation.** You attempted an UPDATE procedure for a record of an indexed file that would have changed the value of a key field, and this situation was disallowed when the file was created.

User Action. If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly change a key field, or recreate the file specifying that the key field is permitted to change. Consult Chapter 4, Input and Output with RMS, for more information.

KEYDEFINC, KEY "nnn" definition is inconsistent with this file

**Explanation.** An indexed file of type RECORD was opened, and the component type contained fields whose KEY attributes did not match those of the existing file. The number of the key in error is displayed in the message.

**User Action.** Correct the RECORD definition so that it describes the correct KEY fields, or recreate the file so that it matches the declared keys. Consult Chapter 4, Input and Output with RMS, for more information.

KEYDUPNOT, key field duplication is not allowed

**Explanation.** You attempted an UPDATE or PUT procedure for a record of an indexed file that would have duplicated a key field value of an existing record, and this situation was disallowed when the file was created.

**User Action.** If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the PUT or UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly duplicate a key field, or recreate the file specifying that the key field is permitted to be duplicated. Consult Chapter 4, Input and Output with RMS, for more information.

KEYNOTDEF, KEY "nnn" is not defined for this file

**Explanation.** You attempted a FINDK or RESETK procedure on an indexed file, and the key number specified does not exist in the file.

**User Action.** Correct the program so that the correct key numbers are used when accessing the file.

KEYVALINC, key value is incompatible with the file's key "nnn"

**Explanation.** The key value specified for the FINDK procedure was incompatible in type or size with the key field of the file.

**User Action.** Make sure that the correct key value is being specified for FINDK. Correct the program so that the type of the key value is compatible with the key of the file. Consult *Programming in VAX PASCAL* for more information.

LINTOOLON, line is too long, exceeded record length by "nnn" character(s)

**Explanation.** You attempted a WRITE, PUT, WRITEV, or other output procedure on a text file that would have placed more characters in the current line than the record length of the file would allow. The number of characters that did not fit is displayed in the message.

**User Action.** Correct the program so that it does not place too many characters in the current line. If appropriate, use the WRITELN procedure, or specify an increased RECORD\_LENGTH parameter when opening the file with the OPEN procedure.

## LINVALEXC, LINELIMIT value exceeded

**Explanation.** The number of lines written to the file exceeded the maximum specified as the line limit. The line-limit value is determined by the translation of the logical name PAS\$LINELIMIT, if any, or the value specified in a call to the LINELIMIT procedure for the file.

**User Action.** As appropriate, correct the program so that it does not write as many lines, or increase the line limit for the file. Note that if a line limit is specified for a nontext file, each PUT procedure called for the file is considered to be one line. Consult *Programming in VAX PASCAL* for more information.

MODNEGNUM, MOD of a negative modulus has no mathematical definition

**Explanation.** In the MOD operation A MOD B, the operand B must have a positive integer value.

**User Action.** Correct the program so that the operand B has a positive integer value.

NEGDIGARG, negative Digits argument to BIN, HEX or OCT is not allowed

**Explanation.** You attempted to specify a negative value for the Digits argument in a call to the BIN, HEX, or OCT procedure, which is not permitted.

**User Action.** Correct the program so that only nonnegative Digits arguments are used for calls to BIN, HEX, and OCT.

NEGWIDDIG, negative Width or Digits specification is not allowed

**Explanation.** A WRITE or WRITEV procedure on a text file contained a field-width specification that included a negative Width or Digits value, which is not permitted.

**User Action.** Correct the program so that only nonnegative Width and Digits parameters are used.

NOTVALTYP, "string" is not a value of type "type"

**Explanation.** You attempted a READ or READV procedure on a text file, but the value read could not be expressed in the specified type. For example, this error results if a real value read is outside the range of the identifier's type, or if an enumerated value is read that does not match any of the valid constant identifiers in its type.

**User Action.** Correct the program or the input data so that the values read are compatible with the types of the identifiers receiving the data.

ORDVALOUT, ordinal value is out of range

**Explanation.** A value of an ordinal type is outside the range of values specified by the type. For example, this error results if you attempt to use the SUCC function on the last value in the type or the PRED function on the first value.

**User Action.** Correct the program so that all ordinal values are within the range of values specified by the ordinal type.

ORGSPEINC, ORGANIZATION specified is inconsistent with this file

**Explanation.** The value of the ORGANIZATION parameter for the OPEN procedure that opened an existing file was inconsistent with the actual organization of the file.

**User Action.** Correct the program so that the correct organization is specified. Consult Chapter 4, Input and Output with RMS, for more information.

## PADLENERR, PAD length error

**Explanation.** The length of the character string to be padded by the PAD function is greater than the length specified as the finished size, or the finished size specified is greater than 65535.

**User Action.** Correct the call to PAD so that the finished size specified describes a character string of the correct length. See *Programming in VAX PASCAL* for the rules governing the PAD function.

# PTRREFNIL, pointer reference to NIL

**Explanation.** You attempted to evaluate a pointer value while its value was NIL.

**User Action.** Make sure that the pointer has a value before you attempt to evaluate it. See *Programming in VAX PASCAL* for more information on pointer values.

RECLENINC, RECORD\_LENGTH specified is inconsistent with this file

**Explanation.** The record length obtained from the file component's length or from the value of the RECORD\_LENGTH parameter specified for the OPEN procedure was inconsistent with the actual record length of an existing file.

**User Action.** Correct the program so that the record length specified, if any, is consistent with the file. Consult Chapter 4, Input and Output with RMS, for more information.

RECTYPINC, RECORD\_TYPE specified is inconsistent with this file

**Explanation.** The value of the RECORD\_LENGTH parameter specified for the OPEN procedure was inconsistent with the actual record type of an existing file.

**User Action.** Correct the program so that the record type specified, if any, is consistent with the file. Consult Chapter 4, Input and Output with RMS, for more information.

RESNOTALL, RESET is not allowed on an unopened internal file

**Explanation.** You attempted a RESET procedure for a nonexternal file that was not open. This operation is not permitted because RESET must operate on an existing file, and there is no information associated with a nonexternal file that allows RESET to open it.

**User Action.** Correct the program so that nonexternal files are opened before using RESET. Either OPEN or REWRITE may be used to open a nonexternal file. Consult *Programming in VAX PASCAL* for more information.

#### REWNOTALL, REWRITE is not allowed for a shared file

**Explanation.** You attempted a REWRITE procedure for a file for which the program did not have exclusive access. REWRITE requires that no other users be allowed to access the file while the file's data is deleted. Note that this message may also be issued if you do not have permission to write to the file.

**User Action.** Correct the program so that the file is opened with SHARING := NONE, which is the default, before performing a REWRITE.

SETASGVAL, set assignment value has element out of range

**Explanation.** You attempted to assign to a set variable a value that is outside the range specified by the variable's component type.

**User Action.** Correct the assignment statement so that the value being assigned falls within the component type of the set variable. See *Programming in VAX PASCAL* for more information on sets.

SETCONVAL, set constructor value out of range

**Explanation.** You attempted to include in a set constructor a value that is outside the range specified by the set's component type or a value that is greater than 255 or less than 0.

**User Action.** Correct the constructor so that it includes only those values within the range of the set's component type. See *Programming in VAX PASCAL* for more information on sets.

## STRASGLEN, string assignment length error

**Explanation.** You attempted to assign to a string variable a character string that is longer than the declared maximum length of the variable (if the variable's type is VARYING) or that is not of the same length as the variable (if the variable's type is PACKED ARRAY OF CHAR).

**User Action.** Correct the program so that the string is of a correct length for the variable to which it is being assigned.

# STRCOMLEN, string comparison length error

**Explanation.** You attempted to compare two character strings that do not have the same current length.

**User Action.** Correct the program so that the two strings have the same length at the time of the comparison.

# SUBASGVAL, subrange assignment value out of range

**Explanation.** You attempted to assign to a subrange variable a value that is not contained in the subrange type.

**User Action.** Correct the program so that all values assigned to a subrange variable fall within the variable's type.

## SUBSTRSEL, SUBSTR selection error

**Explanation.** A SUBSTR function attempted to extract a substring that was not entirely contained in the original string.

**User Action.** Correct the call to SUBSTR so that it specifies a substring that can be extracted from the original string. See *Programming in VAX PASCAL* for complete information on the SUBSTR function.

# TEXREQSEQ, textfiles require sequential organization and access

**Explanation.** You attempted to open a file of type TEXT that either did not have sequential organization, or had an ACCESS\_METHOD other than SEQUENTIAL (the default) when opened by the OPEN procedure.

**User Action.** Make sure that the program refers to the correct file. Correct the program so that only sequential organization and access are used for text files.

## TRUNOTALL, TRUNCATE is not allowed for a shared file

**Explanation.** You attempted to call the TRUNCATE procedure for a file that was opened for shared access. You cannot truncate files that might be shared by other users. This message may also be issued if you do not have permission to write to the file.

**User Action.** Correct the program so that it does not attempt to truncate shared files. If the file is opened with the OPEN procedure, do not specify a value other than NONE (the default) for the SHARING parameter.

# UPDNOTALL, UPDATE not allowed for a sequential organization file

**Explanation.** You attempted to call the UPDATE procedure for a sequential file. UPDATE is only valid on relative and indexed files.

**User Action.** Correct the program so that it does not attempt to use UPDATE for sequential files, or recreate the file with relative or indexed organization. If you are using direct access on a sequential file, individual records can be updated by using LOCATE and PUT procedures. Consult Chapter 4, Input and Output with RMS, to see whether a different file organization may be appropriate for your application.

# VARINDVAL, VARYING index value exceeds current length

**Explanation.** The index value specified for a VARYING string is greater than the string's current length.

**User Action.** Correct the index value so that it specifies a legal character in the string.

# WRIINVENU, WRITE of an invalid enumerated value

**Explanation.** You attempted to write an enumerated value using a WRITE or WRITEV procedure, but the internal representation of that value was outside the possible range for the enumerated type.

**User Action.** Verify that your program is not improperly using PRED, SUCC, or type casting to assign an invalid value to a variable of enumerated type.

# **Errors Returned by the STATUS and STATUSV Functions**

This appendix lists the error conditions detected by the STATUS and STATUSV functions, their symbolic names, and the corresponding values. The symbolic names and their values are defined in the file SYS\$LIBRARY:PASSTATUS.PAS, which you can include with a %INCLUDE directive in a CONST section of your program. To test for a specific condition, you compare the STATUS or STATUSV return values against the value of a symbolic name.

Note that the symbolic names correspond to some of the run-time errors listed in Appendix A, Diagnostic Messages; however, not all run-time errors can be detected by STATUS.

Table B-1 lists the symbolic names and the values returned by the STATUS and STATUSV functions and explains the error condition that corresponds to each value.

Table B-1: STATUS and STATUSV Return Values

Name	Value	Meaning
PAS\$K_ACCMETINC	5	Specified access method is not compatible with this file
PAS\$K_AMBVALENU	30	"String" is an ambiguous value for the enumerated type "type"
PAS\$K_CURCOMUND	73	DELETE or UPDATE was attempted while the current component was undefined
PAS\$K_DELNOTALL	100	DELETE is not allowed for a file with sequential organization
PAS\$K_EOF	-1	File is at end-of-file
PAS\$K_ERRDURCLO	16	Error occurred while the file was being closed
PAS\$K_ERRDURDEL	101	Error occurred during execution of DELETE
PAS\$K_ERRDUREXT	127	Error during EXTEND
PAS\$K_ERRDURFIN	102	Error occurred during execution of FIND or FINDK
PAS\$K_ERRDURGET	103	Error occurred during execution of GET
PAS\$K_ERRDUROPE	2	Error occurred during execution of OPEN
PAS\$K_ERRDURPRO	36	Error occurred during prompting
PAS\$K_ERRDURPUT	104	Error occurred during execution of PUT
PAS\$K_ERRDURRES	105	Error occurred during execution of RESET or RESETK
PAS\$K_ERRDURREW	106	Error occurred during execution of REWRITE
PAS\$K_ERRDURTRU	107	Error occurred during execution of TRUNCATE
PAS\$K_ERRDURUNL	108	Error occurred during execution of UNLOCK
PAS\$K_ERRDURUPD	109	Error occurred during execution of UPDATE
PAS\$K_ERRDURWRI	50	Error occurred during execution of WRITELN

Table B-1: (Cont.) STATUS and STATUSV Return Values

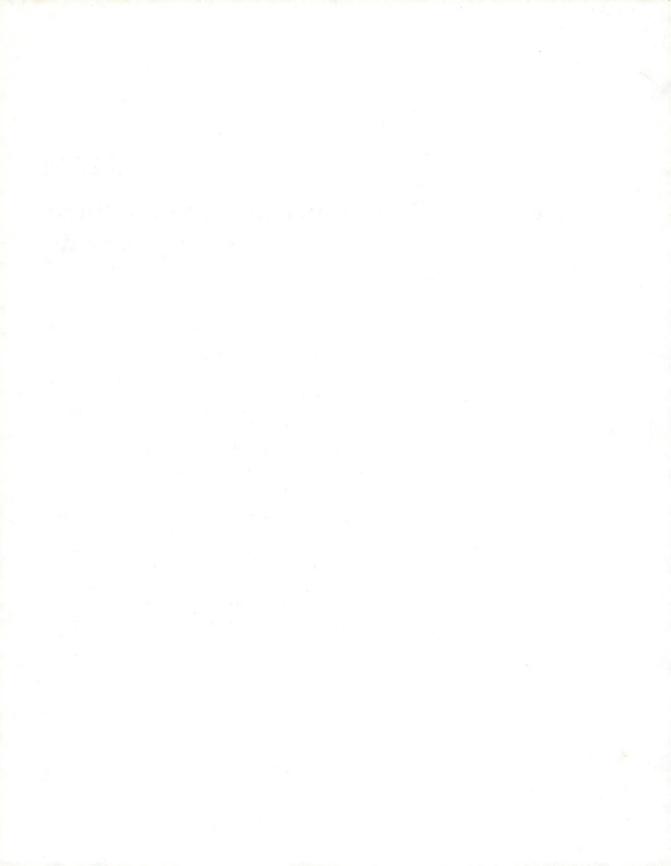
Name	Value	Meaning
PAS\$K_EXTNOTALL	128	EXTEND is not allowed for a shared file
PAS\$K_FAIGETLOC	74	GET failed to retrieve a locked component
PAS\$K_FILALRCLO	15	File is already closed
PAS\$K_FILALROPE	1	File is already open
PAS\$K_FILNAMREQ	14	File name must be specified in order to save, print, or submit an internal file
PAS\$K_FILNOTDIR	110	File is not open for direct access
PAS\$K_FILNOTFOU	3	File was not found
PAS\$K_FILNOTGEN	111	File is not in Generation mode
PAS\$K_FILNOTINS	112	File is not in Inspection mode
PAS\$K_FILNOTKEY	113	File is not open for keyed access
PAS\$K_FILNOTOPE	114	File is not open
PAS\$K_FILNOTSEQ	115	File does not have sequential organizatio
PAS\$K_FILNOTTEX	116	File is not a text file
PAS\$K_GENNOTALL	117	Generation mode is not allowed for a READONLY file
PAS\$K_GETAFTEOF	118	GET attempted after end-of-file has beer reached
PAS\$K_INSNOTALL	119	Inspection mode is not allowed for a WRITEONLY file
PAS\$K_INSVIRMEM	120	Insufficient virtual memory
PAS\$K_INVARGPAS	121	Invalid argument passed to a PASCAL Run-Time Library procedure
PAS\$K_INVFILSYN	4	Invalid syntax for file name
PAS\$K_INVKEYDEF	9	Key definition is invalid
PAS\$K_INVRECLEN	12	Record length nnn is invalid
PAS\$K_INVSYNBIN	37	"String" is invalid syntax for a binary value
PAS\$K_INVSYNENU	31	"String" is invalid syntax for a value of a enumerated type

Table B-1: (Cont.) STATUS and STATUSV Return Values

Name	Value	Meaning
PAS\$K_INVSYNHEX	38	"String" is invalid syntax for a hexadecimal value
PAS\$K_INVSYNINT	32	"String" is invalid syntax for an integer
PAS\$K_INVSYNOCT	39	"String" is invalid syntax for an octal value
PAS\$K_INVSYNREA	33	"String" is invalid syntax for a real number
PAS\$K_INVSYNUNS	34	"String" is invalid syntax for an unsigned integer
PAS\$K_KEYCHANOT	72	Changing the key field is not allowed
PAS\$K_KEYDEFINC	10	KEY(nnn) definition is inconsistent with this file
PAS\$K_KEYDUPNOT	71	Duplication of key field is not allowed
PAS\$K_KEYNOTDEF	11	KEY(nnn) is not defined in this file
PAS\$K_KEYVALINC	70	Key value is incompatible with file's key nnn
PAS\$K_LINTOOLON	52	Line is too long; exceeds record length by nnn character(s)
PAS\$K_LINVALEXC	122	LINELIMIT value exceeded
PAS\$K_NEGWIDDIG	53	Negative value in Width or Digits (of a field width specification) is invalid
PAS\$K_NOTVALTYP	35	"String" is not a value of type "type"
PAS\$K_ORGSPEINC	8	Specified organization is inconsistent with this file
PAS\$K_RECLENINC	6	Specified record length is inconsistent with this file
PAS\$K_RECTYPINC	7	Specified record type is inconsistent with this file
PAS\$K_RESNOTALL	124	RESET is not allowed for an internal file that has not been opened
PAS\$K_REWNOTALL	123	REWRITE is not allowed for a file opened for sharing

Table B-1: (Cont.) STATUS and STATUSV Return Values

Name	Value	Meaning
PAS\$K_SUCCESS	0	Last file operation completed successfully
PAS\$K_TEXREQSEQ	13	Text files must have sequential organization and sequential access
PAS\$K_TRUNOTALL	125	TRUNCATE is not allowed for a file opened for sharing
PAS\$K_UPDNOTALL	126	UPDATE is not allowed for a file that has sequential organization
PAS\$K_WRIINVENU	54	WRITE attempted on an invalid enumerated value



# **Declaring Run-Time Library Procedures** in VAX PASCAL

This appendix describes how to translate the procedure parameter notation used in the VAX/VMS Run-Time Library Routines Reference Manual into procedure and function declarations in VAX PASCAL.

Procedure parameter notation provides a shorthand method for specifying the access type, data type, passing mechanism, and parameter form of each parameter to a Run-Time Library procedure. This notation is also used to describe function values and return statuses.

Procedure parameter notation uses the following elements:

<name>.<access type><data type>.<passing mechanism><parameter form>

These elements are described in this appendix as follows:

- Table C-1 explains the access type codes and shows how to translate them into VAX PASCAL semantics and foreign mechanisms.
- Table C-2 explains the data type codes and shows how they correspond to VAX PASCAL data types.
- Table C-3 explains the passing mechanism codes and shows how to translate them into VAX PASCAL semantics and foreign mechanisms.
- Table C-4 explains the parameter form codes and shows how to translate them into VAX PASCAL data types and foreign mechanisms.

#### Example

The VAX/VMS Run-Time Library Routines Reference Manual describes the parameter to a procedure with the following notation:

seed.mlu.r

You decipher this notation as follows:

seed Procedure parameter name

m Modify access type (semantics)

lu Longword logical data type

By-reference passing mechanism

Scalar parameter form (notation omitted)

The tables in this appendix help you translate the procedure parameter notation in this example into a formal parameter declaration in VAX PASCAL:

Modify access type; in VAX PASCAL, this access type is transm

lated into variable semantics, or into foreign semantics with a

variable as the actual parameter.

lu Longword logical data type; in VAX PASCAL, this type corre-

sponds to type UNSIGNED.

r By-reference passing mechanism; in VAX PASCAL, this mecha-

nism is specified by variable or value semantics (no conformant schemas allowed), by a %REF mechanism specifier on either the formal or the actual parameter, or by a [REFERENCE] attribute

on the formal parameter.

Scalar parameter form (notation omitted); in VAX PASCAL, this

form corresponds to an ordinal or a real type.

By finding the intersection of these requirements, you can determine that the formal parameter must be of type UNSIGNED. The semantics or mechanism for this parameter can be either of two possibilities:

- A VAR parameter of type UNSIGNED
- A variable of type UNSIGNED passed to a formal parameter which is either preceded by a %REF mechanism specifier, or contains the [REFERENCE] attribute.

Table C-1: Access Type Translations

Notation	Meaning	VAX PASCAL Translation
c	Call after stack unwind	Procedure or function parameter passed by a by-immediate value mechanism.
f	Function call (before return)	Function parameter
j	Jump after unwind	Not available
m	Modify	Variable semantics, foreign semantics with a variable as actual parameter
r	Read-only	Value or foreign semantics
S	Call without stack unwind	Procedure parameter
W	Write-only	Variable semantics, foreign semantics with a variable as actual parameter

Table C-2: Data Type Translations

CAL Translation	Meaning	Notation
lue	 Virtual address	a
ble <sup>1</sup>	Absolute date and time	adt
ble	8-bit relative virtual address	arb
ble	32-bit relative virtual address	arl
ble	16-bit relative virtual address	arw
orange of type with a size of 8	Byte integer (signed)	b
ble	Bound label value	blv
or func- neter without D attribute	Bound procedure value	bpv

 $<sup>^{1}\</sup>mathrm{Can}$  be simulated in PASCAL with a record type definition.

Table C-2 (Cont.): Data Type Translations

Notation	Meaning	VAX PASCAL Translation
bu	Byte logical (unsigned)	BOOLEAN, CHAR, enumerated type with 256 or fewer values, unsigned subrange of INTEGER or UNSIGNED with size of 8 bits
c	Single character	CHAR
cit	COBOL intermediate temporary	Not available <sup>1</sup>
ср	Character pointer	^CHAR
d	D_floating	DOUBLE with NOG_ FLOATING attribute or qualifier
dc	D_floating complex	Not available <sup>1</sup>
dsc	Descriptor (used by descriptors)	Not available <sup>1</sup>
f	F_floating	REAL or SINGLE
fc	F_floating complex	Not available
g	G_floating	DOUBLE with G_ FLOATING attribute or qualifier
gc	G_floating complex	Not available <sup>1</sup>
h	H_floating	QUADRUPLE
hc	H_floating complex	Not available <sup>1</sup>
1	Longword integer (signed)	INTEGER
lc	Longword return status	BOOLEAN, INTEGER, condition status record
lu	Longword logical (unsigned)	UNSIGNED

 $<sup>^{1}</sup>$ Can be simulated in PASCAL with a record type definition.

Table C-2 (Cont.): Data Type Translations

Notation	Meaning	VAX PASCAL Translation
nl	Numeric string, left separate sign	PACKED ARRAY[1n] OF CHAR
nlo	Numeric string, left overpunched sign	PACKED ARRAY[1n] OF CHAR
nr	Numeric string, right separate sign	PACKED ARRAY[1n] OF CHAR
nro	Numeric string, right over- punched sign	PACKED ARRAY[1n] OF CHAR
nu	Numeric string, unsigned	PACKED ARRAY[1n] OF CHAR
nz	Numeric string, zoned sign PACKED ARRAY[1n] OF CHAR	
О	Octaword integer (signed)	Not available <sup>1</sup>
ou	Octaword logical (unsigned)	Not available <sup>1</sup>
p	Packed decimal string	Not available <sup>1</sup>
q	Quadword integer (signed)	Not available <sup>1</sup>
qu	Quadword logical (unsigned)	Not available <sup>1</sup>
t	Character-coded text string	PACKED ARRAY[1n] OF CHAR
u	Smallest addressable storage unit	Same as byte logical
v	Aligned bit string	VAR parameter (cannot be a conformant parameter), parameter passed by a byreference mechanism (can be a conformant parameter)

<sup>&</sup>lt;sup>1</sup>Can be simulated in PASCAL with a record type definition.

Table C-2 (Cont.): Data Type Translations

Notation	Meaning	VAX PASCAL Translation
vu	Unaligned bit string	Not available
vt	Varying character-coded text string	VARYING OF CHAR
w	Word integer (signed)	Signed subrange of INTEGER with a size of 16 bits
wu	Word logical (unsigned)	Enumerated type, unsigned subrange of INTEGER or UNSIGNED with size of 16 bits
x	Data type in descriptor	VAR or value conformant schema, any %DESCR parameter
z	Unspecified	%DESCR parameter of a set type (causes descriptor of type DSC\$K_DTYPE_Z to be generated)
zem	Procedure entry mask	Procedure or function parameter passed by a by-immediate value mechanism specifier
zi	Sequence of instruction	Not available

Table C-3: Passing Mechanism Translations

Notation	Meaning	VAX PASCAL Translation
d	By descriptor	VAR or value parameter (can be a conformant parameter), %DESCR, %STDESCR, [CLASS_NCA], or [CLASS_S] parameter
r	By reference	VAR or value parameter (no conformant parameters allowed), parameter which uses %REF or [REFERENCE] (may be a conformant parameter), procedure parameter which uses %IMMED or
		[IMMEDIATE] if required data type is zem
v	By immediate value	Parameter which uses either %IMMED or [IMMEDIATE]

Table C-4: Parameter Form Translations

Scalar	Ordinal or real type
Array reference or descriptor	Array type, conformant array schema, or [CLASS_A]
Dynamic string descriptor	Not available <sup>1</sup>
Noncontiguous array descriptor	[CLASS_NCA]
Procedure reference or descriptor	Procedure parameter
Fixed-length string descriptor	%STDESCR parameter, [CLASS_S]
String with bounds descriptor	Not available
Scalar decimal descriptor	Not available
	Array reference or descriptor  Dynamic string descriptor  Noncontiguous array descriptor  Procedure reference or descriptor  Fixed-length string descriptor  String with bounds descriptor

 $<sup>^{1}\</sup>mbox{Can}$  be simulated in PASCAL with a record type definition and passed by reference.

Table C-4 (Cont.): Parameter Form Translations

Notation	Meaning	VAX PASCAL Translation
uba	Unaligned bit string array descriptor	Not available
ubs	Unaligned bit string descriptor	Not available
ubsb	Unaligned bit string with bounds descriptor	Not available.
vs	Varying string descriptor	Value, VAR, or %DESCR conformant parameter of type VARYING OF CHAR or %DESCR parameter of type VARYING OF CHAR
vsa	Varying string array descriptor	Value, VAR, or %DESCR conformant parameter of type ARRAY OF VARYING OF CHAR, or %DESCR parameter of type ARRAY OF VARYING OF CHAR
x	Class type in descriptor	%DESCR parameter, [CLASS_A], [CLASS_ NCA], or [CLASS_S]
x1	Fixed-length or dynamic string descriptor	%STDESCR parameter of type PACKED ARRAY OF CHAR, or [CLASS_S]

# **Entry Points to PASCAL Utilities**

This appendix describes the entry points to utilities in the VAX Run-Time Library that can be called as external routines by a VAX PASCAL program. These utilities allow you to access VAX PASCAL extensions that are not directly provided by the language.

# D.1 PAS\$FAB(f)

The PAS\$FAB function returns a pointer to the RMS File Access Block (FAB) of file f. After this function has been called, the FAB can be used to get information about the file and to access RMS facilities not explicitly available in the PASCAL language.

The component type of file f can be any type; the file must be open.

PAS\$FAB is an external function that must be explicitly declared by a declaration such as the following:

```
Unsafe_File = [UNSAFE] FILE OF CHAR;
  Ptr_to_FAB = ^FAB$TYPE;
FUNCTION PAS$FAB
   (VAR F : Unsafe _File) : Ptr_to_FAB;
  EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS\$FAB. The type FAB\$TYPE is defined in the VAX PASCAL environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS FAB does not interfere with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the FAB is used, which may in turn require you to change your program. See the VAX/VMS Run-Time Library Routines Reference Manual for more information.

# D.2 PAS\$RAB(f)

The PAS\$RAB function returns a pointer to the RMS Record Access Block (RAB) of file f. After this function has been called, the RAB can be used to get information about the file and to access RMS facilities not explicitly available in the PASCAL language.

The component type of file f can be any type; the file must be open.

PAS\$RAB is an external function that must be explicitly declared by a declaration such as the following:

```
Unsafe_File = [UNSAFE] FILE OF CHAR;
  Ptr_to_RAB = ^RAB$TYPE;
FUNCTION PASSRAB
   (VAR F : Unsafe_File) : Ptr_to_RAB;
   EXTERN:
```

This declaration allows a file of any type to be used as an actual parameter to PAS\$RAB. The type RAB\$TYPE is defined in the VAX PASCAL environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS RAB does not interfere with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the RAB is used, which may in turn require you to change your program. See the VAX/VMS Run-Time Library Routines Reference Manual for more information.

# D.3 PAS\$MARK2(s)

The PAS\$MARK2 function returns a pointer to a heap-allocated object of the size specified by s. If this pointer value is then passed to the PAS\$RELEASE2 function, all objects allocated with NEW or PAS\$MARK2 since the object was allocated are deallocated. PAS\$MARK2 and PAS\$RELEASE2 are provided only for compatibility with some other implementations of PASCAL. Their use is not recommended in a modular programming environment.

PAS\$MARK2 is an external function that must be explicitly declared. Since the parameter to PAS\$MARK2 is the size of the object (unlike the parameter to the predeclared procedure NEW), the best method for using this function is to declare a separate function name for each object you wish to mark. The following example shows how PAS\$MARK2 could be declared and used as a function named Mark\_Integer to allocate and mark an integer variable:

```
TYPE
   Ptr_to_Integer = ^Integer;
VAR
   Marked_Integer: Ptr_to_Integer;
[EXTERNAL(PAS$MARK2)] FUNCTION Mark_Integer
   (%IMMED S : Integer := SIZE(Integer))
   : Ptr_to_Integer;
   EXTERN;
```

Marked\_Integer := Mark\_Integer;

The parameter to PAS\$MARK2 can be 0, in which case the function value is only a pointer to a marker, and cannot be used to store data.

# D.4 PAS\$RELEASE2(p)

The PAS\$RELEASE2 function deallocates all storage allocated by NEW or PAS\$MARK2 since the call to PAS\$MARK2 that allocated the parameter p.

PAS\$MARK2 and PAS\$RELEASE2 are provided only for compatibility with some other implementations of PASCAL. Their use is not recommended in a modular programming environment. PAS\$RELEASE2 disables AST-delivery during its execution, and thus should not be used in a real-time environment.

PAS\$RELEASE2 is an external function that must be explicitly declared. An example of its declaration and use is as follows:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer : Ptr_to_Integer;

[EXTERNAL(PAS$RELEASE2)] PROCEDURE Release
   (P : [UNSAFE] Ptr_to_Integer);
   EXTERN;
   .
   .
   Release (Marked_Integer);
```

In this example, Marked\_Integer is assumed to contain the pointer value returned by a previous call to PAS\$MARK2. See Section D.3 for information about PAS\$MARK2.

# **Differences Between Version 1 and Subsequent Versions**

This appendix describes the differences between VAX PASCAL Version 1 and all subsequent higher versions. In this appendix, the term Version 2+ will refer to both Version 2 and Version 3. The differences between Version 1 and Version 2+ fall into three categories:

- Features that have been decommitted. The previous versions of these features are still supported in Version 2+ to allow you to run existing programs; however, it is recommended that you modify your programs to reflect the new changes.
- Features that are controlled by the /OLD\_VERSION compile-time qualifier.
- Minor changes that are not likely to affect the vast majority of existing VAX PASCAL programs.

#### NOTE

This appendix is intended for users migrating from Version 1 to Version 3. If you are migrating from Version 2 to Version 3, you may disregard the remainder of this appendix as there have been no incompatible changes made between Version 2 and Version 3.

If you modify a program that executed successfully under Version 1 of VAX PASCAL, you should not make changes that conflict with the Version 2+ standard. If conflicts exist and you compile the program with Version 2+, one of two problems may result:

You may get warning messages at compile time.

The program may compile successfully but may not run.

If you must use language features that conflict with Version 2+, you can use the /OLD\_VERSION qualifier at compile time to produce the desired results. The /OLD\_VERSION qualifier and the conflicts that it resolves are described in Section E.2.

# **E.1 Decommitted Features**

The following decommitted features are described in this section:

- Syntax of dynamic array parameters
- Predeclared functions LOWER and UPPER
- Printing of hexadecimal and octal values with the WRITE procedure
- Syntax of the OPEN procedure
- Specification of compiler qualifiers in the source code

# **E.1.1 Dynamic Array Parameters**

Some programming applications require general routines that can process arrays with different bounds. Version 1 of VAX PASCAL allows you to declare routines with dynamic array parameters. You can call the routine with arrays of different sizes, as long as their bounds are within those specified by the formal parameter.

For example, you could write a procedure that sums the components of a one-dimensional array. Each time you use the procedure, you might want to pass arrays with different bounds. Instead of declaring multiple procedures using arrays of each possible size, you could use a dynamic array parameter. The procedure will treat the formal parameter as though its bounds were those of the actual parameter.

In routines that contain dynamic array parameters, you use the predeclared functions LOWER and UPPER to return the lower and upper bounds of the actual array parameter (see Section E.1.2). An array parameter has the following form:

```
array-identifier,...: PACKED ARRAY

[{index-type-identifier},...]

OF type-identifier
```

Note that you must use a type identifier to specify the range of the indexes. You cannot use a subrange. The type identifier can be any of the predefined ordinal types (for example, INTEGER).

The components and indexes of the actual and formal dynamic array parameters must be of compatible types. The rules for dynamic array compatibility are identical to those for compatibility between other arrays, with one exception: the range of the index types of the actual array parameter must be within the range specified for the formal parameter.

The following differences exist between Version 1 and Version 2+ syntax. See Programming in VAX PASCAL for further details on syntax.

- In Version 2+, dynamic arrays are known as conformant arrays, and the syntax that describes them is called a conformant array schema.
- The conformant array schema for Version 2+ requires that the upper and lower bounds of the conformant array parameter be declared with identifiers in the formal parameter list. You can then use these identifiers within the routine block to refer to the upper and lower bounds of the parameter.
- Version 2+ allows the type identifier of a conformant array parameter to be another conformant array schema.

### **E.1.2 LOWER and UPPER Functions**

Version 1 of VAX PASCAL included the predeclared functions LOWER and UPPER, which you could use to determine the upper and lower bounds of dynamic array parameters (see Section E.1.1). Because the syntax of conformant array parameters has changed (see Programming in VAX PASCAL), these functions are no longer necessary. They are supported, however, for programs that use the old syntax. These functions have the form:

LOWER (a [, n]) UPPER (a [ [ n ] ])

The parameter a denotes an array variable; the optional parameter n is an integer constant that denotes a dimension of a. If you omit a value for the parameter n, it defaults to 1. The LOWER function returns the lower bound of the nth dimension of a; the UPPER function returns the upper bound of the nth dimension of a.

# **E.1.3 Printing Hexadecimal and Octal Values**

The following sections explain how to print values in hexadecimal and octal notation using the WRITE procedure. Version 2+ provides the predeclared functions HEX, OCT, and BIN, which return the hexadecimal, octal, and binary equivalents of the input value (see *Programming in VAX PASCAL*). You can use these functions in conjunction with the WRITE, WRITELN, and WRITEV procedures to print values in hexadecimal, octal, and binary notation.

The following formats of the WRITE procedure are used to print hexadecimal and octal values in Version 1.

```
WRITE (expression:field-width HEX,...)
WRITE (expression:field-width OCT,...)
```

#### expression

The value to be written. Arbitrary items (including pointers) can be written in hexadecimal or octal notation to text files.

#### field-width

A positive integer expression indicating the length of the print field.

For hexadecimal values, if the field width specified is less than eight characters, and the output value is greater than the field width, the value being printed is truncated on the left. If the field width is greater than eight characters, and the output value is less than the field width, the field is padded with blanks on the left.

For octal values, if the field width specified is less than 11 characters, and the output value is greater than the field width, the value being printed is truncated on the left. If the field width is greater than 11 characters, and the output value is less than the field width, the field is padded with blanks on the left.

# **Examples**

WRITE (Payroll: 10 HEX);

The value of the variable Payroll is printed in a field of 10 hexadecimal characters.

WRITE (Social\_Security:14 OCT);

The value of the variable Social\_Security is printed in a field of 14 octal characters.

### **E.1.4** The OPEN Procedure

The OPEN procedure opens a file and allows you to specify file parameters. Version 2+ includes new parameters and additional parameter values and has changed some defaults. Table E-1 lists the file parameters available under Version 1, their possible values, and their defaults.

Table E-1: Summary of Version 1 OPEN Parameters

Parameter	Parameter Values	Default
History	OLD or NEW	NEW (OLD, if the file is opened using RESET)
Record-length	Any positive integer	133 bytes
Access-method	DIRECT or SEQUENTIAL	SEQUENTIAL
Record-type	FIXED or VARIABLE	VARIABLE for new files; for old files, record type established at file creation
Carriage-control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, NONE	LIST for all text files; NOCARRIAGE for all other files. Old files use their existing carriage-control parameter

The following differences exist between the Version 1 and Version 2+ OPEN syntax. See Programming in VAX PASCAL for a complete description of the current OPEN procedure.

- In Version 1, the file name is specified as a string constant (VAX/VMS file specification) or a logical name. In Version 2+, a string expression containing a file specification can be used as the file name.
- In Version 2+, the parameter values READONLY and UNKNOWN have been added to the history parameter.
- In Version 2+, the parameter value KEYED has been added to the access-method parameter.
- In Version 2+, the default record type is VARIABLE for new text files and files of type FILE OF VARYING; for all other new files, the default is FIXED. The default for old files remains the same.

- In Version 2+, the default carriage control is LIST for text files and files of type FILE OF VARYING. The default for all other file types and for old files remains the same.
- Version 2+ includes six new parameters for the OPEN procedure: organization, disposition, sharing, user-action, default, and error-recovery. These parameters, their possible values, and their defaults are described in *Programming in VAX PASCAL*.

Note that although direct access to text files is prohibited in both Version 1 and Version 2+, the point at which the error occurs differs. In Version 1, an OPEN procedure is allowed to specify direct access for a text file; the error occurs when a FIND procedure attempts to access the file. In Version 2+, an OPEN procedure that specifies direct access to a text file causes an error to be generated.

# **E.1.5** Specifying Qualifiers in the Source Code

In Version 1 of VAX PASCAL, you could specify compiler qualifiers within comments in the source code. VAX PASCAL Version 2+ does not support this feature. It is recommended that you specify these qualifiers with the PASCAL command when you compile the program. Alternatively, you can use attributes in your program to perform some of the same operations that are performed by compiler qualifiers. For more information, refer to *Programming in VAX PASCAL*.

In Version 1, the CHECK qualifier (abbreviated C) generates code to perform run-time checks. The CROSS\_REFERENCE qualifier (X) produces a cross-reference listing of identifiers. The DEBUG qualifier (D) generates records for the VAX Symbolic Debugger. The LIST qualifier (L) produces a source listing file. The MACHINE\_CODE qualifier (M) includes machine code in the source listing file. The STANDARD qualifier (S) prints informational messages indicating the use of VAX PASCAL extensions. The WARNINGS qualifier (W) prints diagnostics for warning-level errors.

The following syntax indicates how to specify a qualifier(s):

(\*\${qualifier},...[; comment] \*)

#### *aualifier*

A qualifier name or a 1-character abbreviation.

#### comment

The text of a comment, which is optional.

The first character after the comment delimiter must be a dollar sign (\$), which cannot be preceded by a space.

To enable a qualifier, use a plus sign (+) after the qualifier's name or abbreviation. To disable a qualifier, use a minus sign (-) after the qualifier's name or abbreviation. You can specify any number of qualifiers in a single comment. You can also include text in the comment after the qualifiers. The text must be separated from the list of qualifiers by a semicolon.

You can use qualifiers in the source code to enable and disable options during compilation. For example, to generate check code for only one procedure in a program, insert a comment that enables the CHECK qualifier before the procedure declaration. After the end of the procedure declaration, include a comment that disables the qualifier. For example:

(\*\$C+; enable CHECK for TEST1 only \*) PROCEDURE TEST1; END: (\*\$C-; disable CHECK option \*)

You can specify qualifiers in both the source code and the PASCAL command line. Command line qualifiers override source code qualifiers. If, for example, the source code specifies DEBUG+, but you type PASCAL /NODEBUG, the DEBUG option will not be in effect.

# E.2 /OLD\_VERSION Qualifier

The VAX PASCAL standard in early versions conflicts in some respects with that of Version 2+, which is based on Level 0 of the PASCAL standard. The /OLD\_VERSION qualifier on the PASCAL command informs the compiler that it should default to the VAX PASCAL Version 1 standard when conflicts arise. By default, /OLD\_VERSION is disabled so that the compilation conforms to the PASCAL standard.

Because the Version 2+ compiler performs many optimizations on the source code, you should also enable the /NOOPTIMIZE qualifier during the recompilation of old programs. The /NOOPTIMIZE qualifier prevents the compiler from making optimizations that might cause an old program to behave unexpectedly when it is recompiled.

The following sections describe the conflicts between Version 1 and Version 2+ and explain how they are resolved by the /OLD\_VERSION qualifier.

### **E.2.1 Comment Delimiters**

Unlike Version 1, Version 2+, considers the opening comment delimiters (\* and { equivalent; likewise, the closing delimiters \*) and } are considered equivalent. Therefore, a comment begun with (\* can be terminated with }, and a comment begun with { can be terminated by \*).

Recompilation of the program with the /OLD\_VERSION qualifier will cause the Version 1 restriction to be enforced so that you cannot combine comment delimiters in this way.

### E.2.2 %INCLUDE Files

In Version 1 of VAX PASCAL, the default file type of a %INCLUDE file is DAT. However, in Version 2+, the default file type is PAS.

You must use the /OLD\_VERSION qualifier to recompile a program that includes one or more files that have a file type of DAT.

# **E.2.3 Multidimensional Packed Arrays**

Version 1 of the VAX PASCAL compiler interprets the shorthand form of the array type definition

PACKED ARRAY[x,y,z]

to be equivalent to the longer definition

ARRAY[x] OF ARRAY[y] OF PACKED ARRAY[z]

That is, only the last dimension of the array is packed. In Version 2+, however, the shorthand definition above is equivalent to the longer definition:

PACKED ARRAY[x] OF PACKED ARRAY[y] OF PACKED ARRAY[z]

In the Version 2+ interpretation, all dimensions of the array are packed.

You must use the /OLD\_VERSION qualifier to recompile a program that includes a multidimensional packed array of which you want only the last dimension to be packed.

### **E.2.4** Storage of Components

In Version 1 of VAX PASCAL, a component of the subrange type 0..0 in a packed record or array is allocated one bit of storage. In Version 2+, however, a component of this type is not allocated any storage.

You must use the /OLD\_VERSION qualifier to recompile a program in which one bit of storage is required to be allocated for such a component.

#### E.2.5 **Storage of Sets**

In versions of VAX PASCAL, an unpacked set type was always allocated 256 bits. In Version 2+, the allocation size of an unpacked set depends on the set's base type and on whether the unpacked set is allocated in a packed or an unpacked context. See Programming in VAX PASCAL for details.

You must use the /OLD\_VERSION qualifier to recompile a program in which an unpacked set requires an allocation size of 256 bits.

### E.2.6 TEXT Files and FILE OF CHAR

Version 1 of VAX PASCAL considers the predefined file types TEXT and FILE OF CHAR to be equivalent. In Version 2+, however, files of type TEXT are composed of complete lines of characters terminated by an end-of-line marker, while files of type FILE OF CHAR are composed of individual characters. *Programming in VAX PASCAL* provide detailed information about the differences between these two file types.

You must use the /OLD\_VERSION qualifier to recompile a program that requires a TEXT file and a FILE OF CHAR to be treated identically.

### E.2.7 MOD Operator

The MOD operator, as defined by Version 1 of VAX PASCAL, returns the remainder that results from the DIV operation. In Version 2+, however, the MOD operator is equivalent to the mathematical modulus operation. Therefore, Version 2+ allows you to perform the operation I MOD J only when J is a positive number; the MOD function always returns a value from 0 to J–1. To compute the remainder from the DIV operation, Version 2+ provides the REM operator. (See *Programming in VAX PASCAL* for more information about the MOD and REM operators.)

You must use the /OLD\_VERSION qualifier to recompile a program in which you use the MOD operator to compute the remainder.

# **E.2.8** String Variable Parameters to the READ Procedure

In Version 1 of VAX PASCAL, if a READ procedure encounters an end-of-line marker as the first character to be read into a string variable, it ignores the marker and advances the file position to the beginning of the next line of input. In Version 2+, a READ procedure never skips an end-of-line marker that is the first character to be read into a string variable. If a READ procedure encounters an initial end-of-line, the file remains positioned at the end of the line; you must call a READLN procedure to advance the file position to the next line. See *Programming in VAX PASCAL* for further discussion of the READ procedure with string variable parameters.

Recompilation with the /OLD\_VERSION qualifier causes a READ procedure to skip one end-of-line marker that it encounters as the first character to be read into a string variable.

#### E.2.9 **Field Widths**

In Version 1 of VAX PASCAL, a value of type REAL or SINGLE is written with a default field width of 16 characters; a value of type DOUBLE, with 24. In Version 2+, however, the default field width for a value of type REAL or SINGLE is 12 characters; for a value of type DOUBLE, 20.

In addition, Version 1 of VAX PASCAL always expands the field width of a real number written in decimal format (when necessary) so that the real number is preceded by a leading blank. No leading blank is inserted in Version 2+.

You must use the /OLD\_VERSION qualifier to recompile a program in which you want to use the default field width specifications of Version 1.

#### **Global Identifiers** E.2.10

In Version 1 of VAX PASCAL, the names of program-level procedures and functions are considered global identifiers. However, in Version 2+, such names are not considered global unless they have the GLOBAL attribute.

You must use the /OLD\_VERSION qualifier to recompile a program in which the names of program-level routines are to be made global by default.

# **E.2.11** Allocation in Program Sections

Unlike Version 1, Version 2+ of VAX PASCAL does not allocate storage for static, program-level variables in an overlaid program section. (See Programming in VAX PASCAL for information about program section allocation.)

To cause the Version 2+ compiler to treat static, program-level variables and routine identifiers in the same manner as in Version 1, you must use the /OLD\_VERSION qualifier. You can also apply the OVERLAID attribute (see Programming in VAX PASCAL) to a compilation unit to cause the storage for its static, program-level variables to be allocated in an overlaid program section. Enabling /OLD\_VERSION has the same effect as applying the OVERLAID attribute to all compilation units in a program.

# **E.3 Minor Language Changes**

Some minor language changes that were made in Version 1 cannot be controlled by the /OLD\_VERSION qualifier. Such changes, however, are not likely to have adverse affects on most existing VAX PASCAL programs. These changes are as follows:

- To flag language extensions when the /STANDARD qualifier is enabled, Version 2+ uses the PASCAL standard proposed by the International Organization for Standardization as the language definition (standard). The Version 1 language is defined by the PASCAL User Manual and Report by Jensen and Wirth.
- In Version 2+, the /STANDARD qualifier is disabled by default. The Version 2+ compiler does not automatically flag extensions to the PASCAL language definition contained in the ISO standard. /STANDARD was enabled in Version 1.
- Version 2+ ignores all comments whose first character (inside the opening delimiter) is a dollar sign (\$). Note that this behavior prohibits the specification of compile-time qualifiers in the source code, which was legal in Version 1 (see Section E.1.5).
- In Version 2+, the /CHECK qualifier is enabled by default to check the bounds of array and character-string assignments. You can change the default if you wish, and you can also specify the checking of other aspects of your program. /CHECK was disabled by default in Version 1 and did not allow you to specify checking options.
- In Version 2+, a change in the value of the control variable inside the body of a FOR statement does not affect the number of times the loop body is executed. (This behavior is the reverse of Version 1.)
- Version 2+ considers EOLN to be FALSE when EOF is TRUE. (In Version 1, EOLN was TRUE at end-of-file.) This change is necessary to make VAX PASCAL conform to the ISO standard, which forbids a program from testing for EOLN at end-of-file.
- A negative field-width value in a WRITE or WRITELN procedure call generates an error in Version 2+. In Version 1, a negative field-width value defaulted to 0.
- When writing double-precision values, Version 2+ output procedures indicate the exponent by the letter E. (Version 1 used the letter D on output values.) Note, however, that input procedures in both Version 1 and Version 2+ accept either D or E as the exponent letter of double-precision values.

- In Version 2+, the default length of a record in a text file is 132 bytes. Because of an error in Version 1, the default length was actually 254, contrary to the description in the documentation.
- Sets in Version 2+ have different allocation sizes. See Programming in VAX PASCAL for a complete description.
- LIB\$ESTABLISH, the Run-Time Library procedure that sets up condition handlers, cannot be used in Version 2+. Instead, use the new predeclared procedures ESTABLISH and REVERT.
- Run-time condition values have new values in Version 2+. These values are contained in the file SYS\$LIBRARY:PASDEF.PAS. To make them available in your program, include the file in a CONST section.
- In Version 2+, when a nonlocal GOTO statement transfers control from a routine to a labeled statement in an enclosing block, any condition handlers established by intervening routines are called first with the condition SS\$\_UNWIND. In Version 1, a nonlocal GOTO statement transferred control directly to the labeled statement; no condition handlers were executed for intervening routines.
- In Version 2+, you cannot use the predeclared functions SNGL and ORD as function parameters using the Version 1 syntax for function parameter declarations. You must rewrite the formal parameter declarations to use the newer syntax (see Programming in VAX PASCAL).



# **Examples of Using System Services**

The appendix contains examples that involve accessing VAX/VMS system services. It contains sections with examples of:

- Calling RMS procedures
- Synchronizing processes using an AST routine
- Accessing devices using synchronous I/O
- Communicating with other processes
- Sharing code and data
- Gathering and displaying data at terminals
- Creating, accessing, and ordering files
- Measuring and improving performance
- Accessing HELP libraries
- Creating and managing other processes

# **F.1 Calling RMS Procedures**

When you call a Record Management Service, you must supply a value for each parameter. The order of the arguments must correspond with the order shown in the VAX Record Management Services Reference Manual. If you omit an argument, the procedure uses a default value of zero.

The RMS symbolic status codes are defined in the environment file STARLET.PEN, as are some of the RMS routines themselves. STARLET.PEN contains complete formal declarations for the regular file and record processing routines. To use one of these routines in your program, you simply inherit STARLET.PEN and call the routine. The procedure name format for these routines is \$procedure\_name.

A few of the special RMS routines, however, are not declared in STARLET, and must be declared in your program. The procedure name format for these routines is SYS\$procedure\_name. The following program calls the RMS procedure \$SETDDIR to set the default directory for a process.

### Source Program

```
PROGRAM Setddir (OUTPUT):
TYPE
    Word_Integer = [WORD] 0..65535;
VAR
   Dir_Status : INTEGER:
FUNCTION SYS$SETDDIR (
   New_Dir : [CLASS_S] PACKED ARRAY [1..u:INTEGER] OF CHAR;
   Old_Dir_Len : Word_Integer := %IMMED 0;
   Old_Dir : VARYING [lim2] OF CHAR := %IMMED 0):
       INTEGER; EXTERN;
BEGIN
                (* main program *)
   Dir_Status := SYS$SETDDIR ( '[COURSE.PROG.PAS]');
   IF NOT ODD (Dir_Status)
       THEN WRITELN ('Error in SYS$SETDDIR call.')
END.
                (* main program *)
Sample Use:
$ DIRECTORY
                                                             0
Directory DISK$COURSE: [COURSE.PROG.PAS.CALL]
ADDIT.BAS; 1
               DOCOMMAND.PAS; 2 GETINPUT.PAS; 1 GETMSG.PAS; 5
LAB1.PAS;1
              LAB2.PAS:1 LAB3.PAS:1
                                                SETDDIR.PAS;9
SHOWSUM. PAS; 3
Total of 9 files.
$ PASCAL SETDDIR
$ LINK SETTDIR
$ RUN SETTDIR
$ DIRECTORY
```

Directory DISK\$COURSE: [COURSE.PROG.PAS]

CALL.DIR:1	COMU.DIR;1	DEVC.DIR; 1	FILE.DIR;1
HADN . DIR : 1	INTR.DIR; 1	MNAG.DIR;1	PERF.DIR;1
SHAR.DIR; 1	SYNC.DIR; 1	TERM.DIR; 1	PERF.DIR;1

Total of 11 files.

#### Notes:

- 1. The \$SETDIR routine accepts three parameters. In this program we want to omit the second and third arguments, which are optional; therefore we must supply default values in the formal declaration of the routine.
- 2. Before the program is run, the default directory is set to SYS\$DISK:[COURSE.PROG.PAS.CALL] which contains the file SETDDIR.PAS.
- 3. The program is compiled, linked, and executed.
- 4. Another DIRECTORY command shows that the default directory has changed. The current directory is now SYS\$DISK:[COURSE.PROG.PAS].

# F.2 Synchronizing Processes Using an AST Routine

The methods for requesting and declaring an AST procedure are illustrated in the following example.

### **Source Program:**

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Timer_Ast
(INPUT, OUTPUT);
                                                            *)
   ASTPROC.PAS
                                                            *)
    This program sets a 10 second timer which requests
   an AST. The main program then performs arithmetic
                                                            *)
                                                            *)
    operations for the user, interrupted after 10 seconds
                                                            *)
    by the timer AST.
CONST Delay = '0 ::10.00';
TYPE Quadword = RECORD
                 Lo : UNSIGNED;
                 Hi : INTEGER;
                 END:
```

```
VAR Bin_Delay : Quadword;
     Ast_Output: [VOLATILE] TEXT;
     Num1, Num2, Sys_Stat: INTEGER;
(* Declare external RTL routines *)
[ASYNCHRONOUS] FUNCTION LIB$DATE_TIME( VAR Dst_Str:
    VARYING [u] OF CHAR): INTEGER; EXTERN;
[ASYNCHRONOUS] PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER);
    EXTERN:
(* Declare AST procedure *)
[UNBOUND, ASYNCHRONOUS] PROCEDURE Ast_Proc;
   (* This routine is called as an AST procedure. It prints *)
   (* the current time at the terminal.
   VAR Cur_Time : VARYING [80] OF CHAR:
        Time_Stat : INTEGER;
   BEGIN
      Time_Stat:= LIB$DATE_TIME (Cur_Time);
      IF NOT ODD (Time_Stat) THEN LIB$STOP (Time_Stat);
      WRITELN (Ast_Output);
      WRITELN (Ast_Output,'
                              The time is now: ', Cur_Time);
     WRITELN (Ast_Output)
   END; (* Ast_Proc *)
(* Begin main program *)
   OPEN (Ast_Output, 'TT:'); (* output channel for Ast_Proc *)
   REWRITE (Ast_Output);
   (* Convert delay interval to binary format and set timer *)
   Sys_Stat := $BINTIM (Delay, Bin_Delay);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Sys_Stat:= $SETIMR (Daytim:= Bin_Delay, Astadr:= Ast_Proc): 3
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Prompt user for two numbers and multiply them *)
   REPEAT
      WRITELN ('Enter two integers to be multiplied.');
     WRITELN ('(Enter two zeros to quit.)');
     READLN (Num1, Num2);
     WRITELN (' The product is: ', Num1 * Num2)
     UNTIL (Num1 = 0) AND (Num2 = 0);
   WRITELN (' Arithmetic operations completed.')
END.
```

#### Sample Use:

```
$ RUN ASTPROC
Enter two integers to be multiplied.
(Enter two zeros to quit.)
The product is:
                         144
Enter two integers to be multiplied.
(Enter two zeros to quit.)
  The time is now:11-0CT-1984 16:18:43.54
23
45
The product is:
                        1035
Enter two integers to be multiplied.
(Enter two zeros to quit.)
The product is:
Arithmetic operations completed.
```

- 1. AST\_PROC must be declared UNBOUND, because it is passed to \$SETIMR using %IMMED.
- 2. The AST routine needs an output channel to the terminal, separate from the channel used by the main program. The channel is opened and associated to the terminal (TT:) in the main program, because this is a time-consuming operation. You should avoid including unnecessary lengthy operations is AST routines.
  - Any global variables, like AST\_OUTPUT, referenced by a routine declared ASYNCHRONOUS, must be declared VOLATILE. This attribute tells the compiler that the variable is subject to change at any time, and should be optimized carefully.
- 3. The ASTADR parameter in the call to \$SETIMR is the address of the entry point of an AST routine. The AST routine will be executed when the timer expires. If the AST routine is to be executed repeatedly, rather than just once, the timer should be reset at the end of the AST routine.
- 4. When the AST is delivered, the AST routine interrupts the program and outputs this message.

# F.3 Accessing Devices Using Synchronous INPUT/OUTPUT

The following example performs output to a terminal via the \$QIOW system service.

#### **Source Program:**

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Qiow( OUTPUT);
                                                             *)
     This program illustrates the use of the $QIOW system
     service to perform synchronous I/O to a terminal.
CONST Text_String = 'This is from a $QIOW.';
       Terminal = 'TT:';
TYPE
       Word_Integer = [WORD] 0..65535;
       Io_Block = RECORD
                   Io_Stat, Count: Word_Integer;
                   Dev_Info : INTEGER
                   END:
VAR Term_chan : Word_Integer;
     Counter : INTEGER;
     Sys_Stat : INTEGER;
     Iostat_Block : Io_Block;
(* Declare external RTL routine *)
PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER); EXTERN;
BEGIN
   (* Assign the channel number *)
                                                              0
   Sys_Stat := $ASSIGN (Terminal, Term_Chan,,);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Output the message twice *)
   FOR Counter:= 1 TO 2 DO
      BEGIN
         Sys_Stat:= $QIOW (Chan:= Term_Chan,
                           Func: = IO$_WRITEVBLK,
                                                              2
                           Iosb: = Iostat_Block,
                             P1:= Text_String,
                             P2:= LENGTH(Text_String),
                             P4:= 32);
         IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
         IF NOT ODD (Iostat_Block.Io_Stat) THEN
                     LIB$STOP (Iostat_Block.Io_Stat);
      END
             (* for *)
END.
```

### Sample Use:

```
$ RUN QIOW
This is from a $QIOW.
This is from a $QIOW.
```

#### Notes:

 TERM\_CHAN receives the channel number from the \$ASSIGN system service.

The process permanent logical name SYS\$OUTPUT is assigned to your termianl when you login. The \$ASSIGN system service will translate the logical name to the actual device name.

2. The function IO\$\_WRITEVBLK requires values for parameters P1, P2, and P4. P1 is the starting address of the buffer containing the message. P2 is the number of bytes to be written to the terminal (in this example, 21). P4 is the carriage control specifier; 32 indicates single space carriage control.

\$OIOW is used, instead of \$QIO, to insure that the output operation will complete before the program terminates.

# **F.4 Communicating with Other Processes**

The following example shows how to create a global section and use it to transfer data between two processes. This example requires that you run programs GLOBAL1 and GLOBAL2 in the same UIC group. GLOBAL1 creates and maps the section, but GLOBAL2 only maps the section. Therefore, GLOBAL1 must be run first.

### **Source Program Number 1:**

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Global1(OUTPUT);
                                                         *)
    GLOBAL1.PAS
   This program creates and maps a global section.
                                                         *)
    Data in the section file is accessed through an
     array.
                                         (* for addresses *)
TYPE Int_Pointer = ^INTEGER;
    File_Type = FILE OF INTEGER;
     Word_Integer = [WORD] 0..65535;
```

```
VAR My_Adr, Sys_Adr: ARRAY [1..2] OF Int_Pointer;
     Iarray: [VOLATILE, ALIGNED(9)] ARRAY [1..50] OF INTEGER;
     Sec_Chan: UNSIGNED;
     Name:
               PACKED ARRAY [1..4] OF CHAR;
     Sec_File: File_Type;
     Sec_Flags: INTEGER;
     Sys_Stat, Counter: INTEGER;
(* Declare user_action open function *)
FUNCTION Get_Chan (
    VAR File_Fab: FAB$TYPE; VAR File_Rab: RAB$TYPE;
    VAR A_File: File_Type): INTEGER;
   (* This routine is called as a user_action function *)
   (* by the OPEN statement. The channel number for
                                                        *)
   (* file is stored in the global variable SEC_CHAN.
                                                        *)
   VAR Open_Status: INTEGER;
  BEGIN
   (* Define appropriate FAB fields *)
  File_Fab.FAB$V_CBT := FALSE;
  File_Fab.FAB$V_CTG := TRUE;
  File_Fab.FAB$V_UFO := TRUE;
  File_Fab.FAB$L_ALQ := 1;
   (* Open the file *)
   Open_Status:= $CREATE (Fab:=File_Fab);
   IF ODD (Open_Status)
    THEN
        BEGIN
         (* get channel number *)
                                                          3
         Sec_Chan:= File_Fab.FAB$L_STV
   Get_Chan:= Open_Status
   END:
             (* get_chan *)
(* Declare external RTL routine *)
```

```
PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER); EXTERN;
   (* Associate with common cluster MYCLUS *)
   $ASCEFC (Efn:= 64, Name:= 'MYCLUS');
   (* Open the file *)
   OPEN (File_Variable:= Sec_File, File_Name:= 'SECTION.DAT',
         History:= New, User_Action:= Get_Chan);
   (* Obtain starting and ending addresses of the section *)
   My_Adr[1]:= ADDRESS (Iarray[1]);
   My_Adr[2]:= ADDRESS (Iarray[50]);
                                                           4
   (* Create and map the temporary global section *)
   Name: = 'GSEC';
   Sec_Flags:= SEC$M_GBL + SEC$M_WRT + SEC$M_DZRO;
   Sys_Stat:= $CRMPSC (My_Adr, Sys_Adr,, Sec_flags, Name,,,
                       %IMMED Sec_Chan, 1,,,);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Write data to the global section *)
   FOR counter:= 1 TO 50 DO
      Iarray[Counter] := Counter;
   (* Write the pages to the file *)
                                                           6
   Sys_Stat := $UPDSEC (Inadr:= Sys_Adr, Efn:= 1);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Wait unit1 pages have been written back to the disk *)
   Sys_Stat := $WAITFR (Efn:= 1);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Wait for GLOBAL2 to update the section *)
   Sys_Stat:= $SETEF (Efn:= 72);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   WRITELN ('Waiting for GLOBAL2 to update section.');
   Sys_Stat:= $WAITFR (Efn:= 73);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Print the modified pages *)
   WRITELN( 'Modified data in the global section: ');
   FOR Counter:= 1 TO 50 DO
      BEGIN
         WRITE (Iarray[Counter]:5);
         IF (Counter REM 10) = 0
             THEN WRITELN:
      END
END.
```

### **Source Program Number 2:**

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Global2( OUTPUT);
                                                             *)
    GLOBAL2.PAS
                                                             *)
    This program maps and modifies a global section
(*
(* after GLOBAL1 creates it. Programs GLOBAL1 and
                                                             *)
                                                             *)
    GLOBAL2 synchronize the processing of the global
                                                             *)
   section via common event flags.
                                         (* for addresses *)
TYPE Int_Pointer = ^INTEGER;
    File_Type = FILE OF INTEGER;
VAR My_Adr: ARRAY [1..2] OF Int_Pointer;
     Iarray: [VOLATILE, ALIGNED(9)] ARRAY [1..50] OF INTEGER;
     Sys_Stat, counter: INTEGER;
PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER); EXTERN;
BEGIN
   (* Obtain starting and ending addresses of the section *)
   My_Adr[1]:= ADDRESS (Iarray[1] );
   My_Adr[2]:= ADDRESS (Iarray[50]);
   (* Associate with common cluster MYCLUS, and wait for
                                                            *)
                                                            *)
   (* event flag to be set.
   Sys_Stat:= $ASCEFC (Efn:= 64, Name:= 'MYCLUS' );
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Sys_Stat:= $WAITFR (Efn:= 72);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Map global section, using WRITE flag *)
   sys_Stat:= $MGBLSC (Inadr:= My_Adr, Flags:= SEC$M_WRT,
                       Gsdnam: = 'GSEC'):
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Output data in global section, and double each value *)
   WRITELN ('Original data in global section:');
   FOR Counter:= 1 TO 50 DO
      BEGIN
         WRITE (Iarray[Counter]:5);
         IF (Counter REM 10) = 0
            THEN WRITELN:
            Iarray[Counter] := Iarray[Counter] * 2
      END;
   (* Set event flag to allow GLOBAL1 to continue *)
   Sys_Stat:= $SETEF (Efn:= 73);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
END.
```

#### Sample Use:

```
$ ! FROM ONE PROCESS
$ RUN GLOBAL1
Waiting for GLOBAL2 to update secton.
Modified data in the global section:
       4 6 8 10 12 14 16 18
                                                   20
   2
   22
        24 26 28 30 32 34 36 38
                                                  40
       44 46 48 50 52 54 56 58
64 66 68 70 72 74 76 78
84 86 88 90 92 94 96 98
                                                   60
   42
   62
                                                   80
   82
                                                  100
$ ! FROM ANOTHER PROCESS
$ RUN GLOBAL2
Original data in global section:
                                    7 8
    1 2 3
                  4 5 6
                                                   10
      12 13 14 15 16 17 18 19
22 23 24 25 26 27 28 29
32 33 34 35 36 37 38 39
42 43 44 45 46 47 48 49
   11
                                                   20
   21
                                                   30
   31
                                                   40
```

- The \$CRMPSC system service maps pages starting at page boundaries. The ALIGN attribute, with a value of 9, is used to ensure that IARRAY starts on a page boundary. (If the binary address of the first element of IARRAY ends in 9 zeros, then it is page aligned.)
- 2. In order to map a disk file as a section, we need a channel for the file. A channel is assigned by the OPEN statement, and the user-action feature of the OPEN statment allows you to obtain the number of the assigned channel.
- 3. Because the user-file-open option (FAB\$V\_UFO) was specified in the file FAB, the file channel number is stored in the file FAB at offset FAB\$L\_STV.
- 4. The starting and ending process virtual addresses of the section are placed in MY\_ADR. The output argument SYS\_ADR receives the starting and ending system virtual addresses. The flag SEC\$M\_ GLOBAL requests a global section. The flag SEC\$M\_WRT indicates that the pages should be writeable as well as readable. The SEC\$M\_ DZRO flag requests pages filled with zeros.
- 5. Data is placed in the global section and then the section is updated with the \$UPDSEC system service. The \$UPDSEC system service executes asynchronously. Therefore, event flag 1 is used to synchronize completion of the update operation.

6. GLOBAL2 maps the existing section as writeable by specifying the SEC\$M\_WRT flag. Note that the SEC\$M\_DZRO flag is not set, since that would destroy any data that might be in the section.

# F.5 Sharing Code and Data

The program called SHAREDFIL is used to update records in a relative file. SHARING := READWRITE is specified on the OPEN statement to invoke the RMS file sharing facility. In this example, the same program is used to access the file from two processes.

#### **Source Program:**

```
PROGRAM Sharedfil (INPUT, OUTPUT);
                                                         *)0
   SHAREDFIL . PAS
    This program can be run from two or more processes
                                                         *)
(* to illustrate the use of an RMS shared file to
                                                         *)
   share data. The program requires the existence of
                                                         *)
                                                         *)
   a relative file named REL.DAT.
CONST Prompt= 'Record number (CTRL/Z to quit): ';
                                                           2
       %INCLUDE 'SYS$LIBRARY: PASSTATUS. PAS'
TYPE Char_Array = PACKED ARRAY [1..10] OF CHAR;
VAR Rel_File: FILE OF Char_Array;
     Rec_Num : INTEGER;
PROCEDURE LIB$STOP (%IMMED Cond_Value: INTEGER); EXTERN;
PROCEDURE Process_Rec;
   (* Output a record and modify value, if necessary *)
   (* global variables: rec_num, rel_file
   BEGIN
     WRITELN( 'file record is: ', Rel_File^);
      (* Request updated record *)
      WRITE( 'New Value or CR: ');
     IF NOT EOLN( INPUT)
         THEN
            BEGIN
                     (* read new information into file buffer *)
           READLN (Rel_File^);
           UPDATE (Rel_File);
            IF STATUS (Rel_File) <> 0
               THEN LIB$STOP( STATUS (Rel_File))
            END (* if not <CR> *)
         ELSE READLN (* read past the <CR> *)
   END; (* process_rec *)
```

```
BEGIN
        (* main *)
   OPEN (FILE_VARIABLE:= Rel_File, FILE_NAME:= 'REL.DAT',
        HISTORY: = Old,
                                   ACCESS_METHOD:= Direct,
         ORGANIZATION: = Relative, SHARING: = Readwrite);
   WRITE (Prompt);
   WHILE NOT EOF (Input) DO
      BEGIN
      (* Get record *)
      READLN (Rec_Num);
      FIND (Rel_File, Rec_Num, ERROR:= CONTINUE);
      CASE STATUS (Rel_File) OF
         PAS$K_SUCCESS: IF NOT UFB (Rel_File)
               THEN Process_Rec
               ELSE WRITELN ('Did not find record ', Rec_Num);
         PAS$K_FAIGETLOC: WRITELN ('Record locked.');
         PAS$K_ERRDURFIN: WRITELN ('Error during FIND.');
         OTHERWISE (* signal the error *)
          LIB$STOP( STATUS (Rel_File));
         END; (* end case *)
      WRITE (Prompt)
      END (* while *)
END.
Sample Use:
* PASCAL SHAREDFIL
$ LINK SHAREDFIL
* RUN SHAREDFIL
Record number (CTRL/Z to quit): 2
file record is: MSPIGGY
New Value or CR: FOZZIE
Record number (CTRL/Z to quit): 1
file record is: KERMIT
New Value or CR:
Record number (CTRL/Z to quit): ^Z
$ RUN SHAREDFIL
Record number (CTRL/Z to quit): 2
Record locked.
Record number (CTRL/Z to quit): 2
Record locked.
Record number (CTRL/Z to quit): 2
file record is: FOZZIE
New Value or CR: MSPIGGY
Record number (CTRL/Z to quit): ^Z
$
```

#### Notes:

- 1. This program requires a relative file named REL.DAT.
- 2. The PASSTATUS.PAS file is included to define the symbolic status codes for the conditions detected by the STATUS function. Notice that no semi-colon is necessary to end the INCLUDE clause.
- 3. SHARING := READWRITE is specified on the OPEN statement to indicate that the file can be shared. Since manual locking was not specified, RMS will automatically control access to the file.
- 4. The second process is not allowed to access record number 2 while the first process is accessing it.
- 5. Once the first process has finished with record number 2, the second process can update it.

# F.6 Gathering and Displaying Data at Terminals

The following example illustrates how to clear a screen, move the cursor to specific locations on the screen, and send and receive data from the terminal using the terminal-independent screen formatting routines.

### **Source Program Number 1:**

```
PROGRAM Usescreen (INPUT, OUTPUT);
(*
       USESCREEN . PAS
       This program shows the use of the RTL screen package. *)
TYPE Word_Integer = [WORD] 0..65535;
VAR Screen_Stat : INTEGER;
   Name : VARYING [20] OF CHAR;
   Put_Flag : Word_Integer;
(* Declare external RTL routines *)
FUNCTION LIB$ERASE_PAGE (Line_No: Word_Integer;
          Col_No: Word_Integer): INTEGER; EXTERN;
FUNCTION LIB$SET_CURSOR (Line_No: Word_Integer;
          Col_No: Word_Integer): INTEGER; EXTERN;
FUNCTION LIB$GET_SCREEN (
          VAR Input_Text: VARYING [lim1] OF CHAR;
          Prompt_Str : VARYING [lim2] OF CHAR;
          VAR Out_Len : Word_Integer := %IMMED 0):
          INTEGER; EXTERN;
```

```
FUNCTION LIBSPUT_SCREEN(
          Out_Text: VARYING [c] OF CHAR;
          Line_No: Word_Integer; Col_No: Word_Integer;
          Flags: Word_Integer): INTEGER; EXTERN;
PROCEDURE LIB$STOP (%IMMED Cond_Value: INTEGER); EXTERN;
(* Begin main program *)
   (* Clear entire screen starting at (1,1) *)
                                                              0
   Screen_Stat:= LIB$ERASE_PAGE( 1,1);
  IF NOT ODD (Screen_Stat) THEN LIB$STOP (Screen_Stat);
   (* Shift cursor position to (10, 20) *)
   Screen_Stat:= LIB$SET_CURSOR(10, 20);
   IF NOT ODD (Screen_Stat) THEN LIB$STOP (Screen_Stat);
   (* Ask user for name using prompt string *)
   Screen_Stat: = LIB$GET_SCREEN (Name,
                            'Please enter your name: ');
   IF NOT ODD (Screen_Stat) THEN LIB$STOP (Screen_Stat);
   (*Display name at (15,30) in reverse video *)
  Put_Flag:= 2;
                   (* that means reverse video *)
   Screen_Stat:= LIB$PUT_SCREEN (Name, 15, 30, Put_Flag);
   IF NOT ODD (Screen_Stat) THEN LIB$STOP (Screen_Stat);
END.
```

No sample run is included for this example because the program requires a video terminal to execute properly.

- 1. The ERASE\_PAGE routine erases text starting at the specified line and column. To erase the entire screen, start at line 1, column 1.
- 2. The cursor may be positioned to any location on the screen, without erasing any information that exists on the screen. Therefore, the cursor can be moved in any direction before displaying or receiving data.
- Data entered by the user can be placed in a buffer specified by the first argument to GET\_SCREEN. Note that no prompt is required, and may not be specified for applications that display an entire form first, and then return to specific fields to obtain data (i.e., treat form as "prompt" for first field read).
- Data may be sent to any location on the screen. For some terminals the special flags attribute can help draw the user's attention to the data; for instance, using reverse-video on a VT100-class terminal. Note that some of the attributes (like blink and bold) require that a VT100 have the advanced-video otpion. Reverse-video is available on all VT100s.

#### **Source Program Number 2:**

This program illustrates the use of the \$QIO system service to communicate with the terminal, and to respond to a CTRL/C typed by a user. It also illustrates the use of function code modifiers.

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Controlc
(INPUT, OUTPUT);
    CONTROLC . PAS
    This program illustrates the use of $QIOs to the
(*
    terminal and establishing an AST routine to
    handle CTRL/C's.
TYPE Word_Integer = [WORD] 0..65535;
     Io_Status = RECORD
                 Io_Stat, Count: Word_Integer;
                 Device_Info : INTEGER
                 END:
                   PACKED ARRAY [1..48] OF CHAR:
VAR Prompt:
    Buffer:
                   VARYING [80] OF CHAR;
              [VOLATILE] Word_Integer;
    Tt_Chan:
    Ast_Output: [VOLATILE] TEXT;
Lo Func: Word Integer:
    Io_Func:
                   Word_Integer;
    Iostat_Block: [VOLATILE] Io_Status;
    Counter:
                   INTEGER;
    Sys_Stat:
                   INTEGER:
[ASYNCHRONOUS] PROCEDURE LIB$STOP (
                         %IMMED Cond_Value: INTEGER); EXTERN;
[ASYNCHRONOUS, UNBOUND] PROCEDURE Cast (Channel: Word_Integer);
   (* This procedure is called as a CTRL/C ast routine. *)
  VAR Cancel_Stat: INTEGER;
  BEGIN
      (* Cancel all I/O on terminal channel *)
      Cancel_Stat:= $CANCEL (Channel);
      IF NOT ODD (Cancel_Stat) THEN LIB$STOP (Cancel_Stat);
      (* Tell user that CTRL/C AST routine was entered *)
      WRITELN (Ast_Output, 'You typed a CTRL/C')
  END:
          (* procedure cast *)
```

```
(* Begin main program *)
BEGIN
   OPEN (Ast_Output, 'TT:');
                                 (* output channel for cast *)
   REWRITE (Ast_Output);
   Prompt:='You have 5 seconds to enter data, or type CTRL/C';
   (* Assign channel to terminal *)
   Sys_Stat:= $ASSIGN (Devnam:= 'SYS$COMMAND', Chan:= Tt_Chan);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Enable CTRL/C AST *)
                                                           ഒ
   Io_Func:= IO$_SETMODE + IO$M_CTRLCAST;
   Sys_Stat:= $QIOW (Chan:= Tt_Chan, Func:= Io_Func,
                     Iosb:=Iostat_Block.
                     P1:= %IMMED Cast, P2:= %REF Tt_Chan);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   IF NOT ODD (Iostat_Block.Io_Stat) THEN
               LIB$STOP (Iostat_Block.Io_Stat);
   (* Issue a timed read, without echo, with a prompt *)
   Io_Func:= IO$_READPROMPT + IO$M_NOECHO + IO$M_TIMED;
   Sys_Stat:= $QIOW (,Tt_Chan, Io_Func, Iostat_Block,,,
                     Buffer.Body, SIZE (Buffer.Body), 5,,
                     %REF Prompt, SIZE (Prompt));
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Buffer.Length: = Iostat_Block.Count;
   (* Check status of I/O and issue appropriate message *)
   IF Iostat_Block.Io_Stat = SS$_NORMAL
      THEN WRITELN ('You typed: ', Buffer)
      ELSE
         IF Iostat_Block.Io_Stat = SS$_CONTROLC
                                                           6
            THEN WRITELN ('Request was cancelled')
            ELSE
              IF Iostat_Block.Io_Stat = SS$_TIMEOUT
                  THEN WRITELN ('You did not respond in time')
                  ELSE WRITELN ('Unexpected status: ',
                                 Iostat_Block.Io_Stat)
END.
Sample Use:
$ RUN CONTROLC
You have 5 seconds to enter data, or type CTRL/C
You typed: HI THERE
$ RUN CONTROLC
You have 5 seconds to enter data, or type CTRL/C
You typed a CTRL/C
Request was cancelled
* RUN CONTROLC
You have 5 seconds to enter data, or type CTRL/C
You did not respond in time
```

- 1. Because CAST is called asynchronously:
  - The procedure must be declared ASYNCHRONOUS and UNBOUND.
  - LIB\$STOP, which is called from the procedure, must also be declared ASYNCHRONOUS.
  - TTCHAN must be declared VOLATILE.
  - A separate output channel to the terminal (ast\_output) must be established.
- 2. Before any \$QIO can be issued, a channel must be assigned to the device.
- 3. A CTRL/C AST routine is declared.
  - Note that this routine will only be entered for the first CTRL/C the user types.
  - The formal parameter for P1 specifies that it will be passed %REF. In this case, you are passing the entry point for a routine, which should be passed %IMMED. The foreign mechanism specifier is used on the actual parameter for P1 to override the formal definition.
  - The %REF mechanism specifier is used on the actual parameter for P2 to override the formal definition of %IMMED. The CAST procedure expects this parameter to be passed to it by reference.
- 4. This request prompts a user for input. It sets up a time limit for waiting between characters typed (five seconds). Also, any characters typed will not be echoed on the terminal. An I/O status block is established to contain the final I/O status and byte count for characters read.
  - The %REF specifier is used on the actual P5 parameter to override type checking against the formal parameter definition, and to pass the parameter by reference rather than %IMMED.
- 5. The LENGTH field of the varying string BUFFER is filled in manually because it is not done by the system service. System services can only accept (and manipulate) fixed-length strings.
  - BUFFER could be defined as a fixed-length string, but we would have to know (at compile-time) the exact size of the string to be read by the \$QIOW.
- 6. Specific tests are made for CTRL/C or timeout detection using the final I/O status in the I/O Status Block.

- 7. The CTRL/C routine cancels any outstanding requests on the terminal channel.
- 8. Before exiting, the CTRL/C AST routine issues a message to the user, indicating that the AST routine was called.

The sample run illustrates the three possibilities that may occur when running this program (entering data, not responding in time, and typing a CTRL/C ).

# F.7 Creating, Accessing, and Ordering Files

The following example illustrates the use of a user\_action function to provide an I/O timeout capability. This capability is not available through the OPEN statement; you must use a user\_action function to set the proper fields in the RAB.

In this program, the user is prompted for information which must be typed within 10 seconds. If no information is entered within that time, the operation will time out and the program will print a message.

### Source Program Number 1:

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Timeout
(INPUT, OUTPUT);
    TIMEOUT. PAS
(* This program illustrates the use of a user-action
(* routine to initialize the I/O time-out period for
(* a terminal. It also illustrates the use of the
                                                         *)
(* STATUS and PAS$RAB functions to obtain the status
                                                        *)
    I/O operations.
CONST %INCLUDE 'SYS$LIBRARY: PASSTATUS. PAS/NOLIST'
VAR
      Term_File: TEXT;
       Number : INTEGER;
       Prompt : PACKED ARRAY [1..33] OF CHAR
               := 'Enter an integer (zero to quit): ';
```

```
PROCEDURE LIB$STOP (%IMMED Cond_Value: INTEGER); EXTERNAL;
(* Declare user_action open function *)
                                                          0
FUNCTION Set_Timeout (
     VAR File_Fab: FAB$TYPE; VAR File_Rab: RAB$TYPE;
     VAR A_File : TEXT): INTEGER;
   (* This routine is called as a user_action function
   (* by the OPEN statement. The file is opened with
                                                        *)
   (* an input time-out period of 10 seconds.
                                                        *)
  VAR Open_Status: INTEGER;
   BEGIN
      (* Set up RAB for TMO of 10 seconds *)
      File_Rab.RAB$B_TMO:= 10;
      File_Rab.RAB$V_TMO:= TRUE;
      (* Set up RAB for RMS prompting *)
      File_Rab.RAB$V_PMT:= TRUE;
      File_Rab.RAB$L_PBF:= IADDRESS (Prompt);
      File_Rab.RAB$B_PSZ:= SIZE (Prompt);
      (* Open the file and connect the record stream *)
      Open_Status:= $CREATE (Fab:=File_Fab);
      IF ODD (Open_Status)
        THEN $CONNECT (Rab:=File_Rab);
      Set_Timeout:= Open_Status
   END;
             (* set_timeout *)
PROCEDURE Process_Get_Error;
   (* Determines the RMS error that caused a PAS$K_ERRDURGET *)
   (* Global variable: term_file *)
   TYPE Rab_Ptr = ^RAB$TYPE:
  VAR Rab_Start: Rab_Ptr;
  FUNCTION PAS$RAB (VAR F: [UNSAFE] TEXT): Rab_Ptr; EXTERN;
  PROCEDURE LIB$SIGNAL (%IMMED Sigargs:
                         [LIST, UNSAFE] INTEGER); EXTERN;
  BEGIN (* process_get_error *)
      Rab_Start:= PAS$RAB (Term_File);
                                                          9
      IF Rab_Start^.RAB$L_STS = RMS$_TMO
         THEN WRITELN ('No response in 10 seconds...')
         ELSE LIB$SIGNAL (Rab_Start^.RAB$L_STS.
                          Rab_Start^.RAB$L_STV)
  END:
          (* procedure *)
```

```
(* Begin main program *)
BEGIN
  OPEN (File_Variable:= Term_File, File_Name:= 'TT:',
         History:= New, User_Action:= Set_Timeout);
  RESET (Term_File);
   (* Accept input until zero entered *)
   Number: = 1;
   WHILE (Number <> 0) DO
     BEGIN
         READ (Term_File, Number, ERROR:= CONTINUE);
         (* Display input and test for errors *)
         CASE STATUS (Term_File) OF
           PAS$K_SUCCESS:
               WRITELN ('You entered: ', Number);
            PAS$K_ERRDURGET:
               Process_Get_Error;
            OTHERWISE
              LIB$STOP (STATUS (Term_File))
            END:
                 (* case *)
         (* clear the rest of the line *)
         WHILE NOT EOLN (Term_File) DO
           GET (Term_File);
       END: (* while not zero *)
END.
Sample Use:
$ PASCAL TIMEOUT
$ LINK TIMEOUT
$ RUN TIMEOUT
Enter an integer (zero to quit): 10
You entered:
Enter an integer (zero to quit):
No response in 10 seconds...
Enter an integer (zero to quit): 0
You entered:
```

- 1. This INCLUDE directive defines the PASCAL error codes.
- 2. Three parameters are passed to a user action routine: the FAB, the RAB, and the file variable. Recall that FAB\$TYPE and RAB\$TYPE are defined in STARLET.PEN.
- 3. All relevant parameters are included in the OPEN statement. The user action routine only alters RMS fields that could not be specified in the OPEN statement.
- 4. The appropriate RAB fields are set to define a timeout period.

5. When a TEXT file is opened with a user action routine, the usual PASCAL prompting feature is not enabled. In other words, though READs and WRITEs are going to the same device, in this example, the following two statements do not display the prompt at the terminal:

```
WRITE (Prompt);
READLN (Term_File, Number);
```

Instead, RMS prompting is enabled for the file by setting the appropriate field in the RAB. Now any READ or GET operation on the file translates to a "read with prompt."

- 6. Once the appropriate fields are set, the file is created and a record stream is connected.
- 7. The program accesses the file using a normal READ statement. When the ERROR parameter is used, it must be specified using keyword calling syntax. In addition, it must be the last parameter in the list.
- 8. The STATUS function returns the status code of the last operation on the file. If the status code is PAS\$K\_ERRDURGET, procedure process\_get\_error is called.
- 9. PASCAL only informs the program that an error occurred during the READ (or GET). To determine the actual RMS error code, procedure process\_get\_error calls PAS\$RAB to obtain a pointer to the file \$AB. The RAB\$L\_STS filed in the RAB contains the actual RMS status code.

If the RMS status code is RMS\$TMO (timeout), the program informs the user. Any other error is unexpected, so the program signals for a condition handler.

### **Source Program Number 2:**

```
PROGRAM Relative (INPUT, OUTPUT);
(* RELATIVE.PAS
                                                          *)
   This program illustrates accessing a relative file
                                                          *)
(* randomly. It also performs some I/O status checks.
                                                          *)
CONST Prompt= 'Record number (CTRL/Z to quit): ';
      %INCLUDE 'SYS$LIBRARY: PASSTATUS. PAS'
TYPE Char_Array = PACKED ARRAY [1..20] OF CHAR;
VAR Rel_File: FILE OF Char_Array;
    Rec_Num : INTEGER:
```

```
PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER); EXTERN;
BEGIN
        (* main program *)
  OPEN (Rel_File, 'REL.DAT', HISTORY:= OLD,
         ORGANIZATION: = RELATIVE, ACCESS_METHOD: = DIRECT);
   (* Get records by record number until e-o-f *)
   WRITE (Prompt);
   WHILE NOT EOF (Input) DO
      BEGIN
      (* Get record *)
      READLN (Rec_Num);
                                                             8
      FIND (Rel_File, Rec_Num, ERROR: = CONTINUE);
      CASE STATUS (Rel_File) OF
         PAS$K_SUCCESS: IF NOT UFB (Rel_File)
               THEN WRITELN (Rel_File^)
               ELSE WRITELN ('Did not find record ',Rec_Num);
         PAS$K_ERRDURFIN: WRITELN ('Error during FIND.');
         OTHERWISE (* signal the error *)
            LIB$STOP (STATUS (Rel_File));
         END; (* end case *)
      WRITE (Prompt)
      END (* while *)
END.
Sample Use:
$ RUN RELATIVE
Record number (CTRL/Z to quit): 7
08001FLANJE119PL1920
Record number (CTRL/Z to quit): 1
07672ALBEHA210SE2100
Record number (CTRL/Z to quit): 30
Did not find record
Record number (CTRL/Z to quit): ^Z
```

- 1. The INCLUDE directive defines the PASCAL status codes.
- 2. This statement defines the file and record processing characteristics. Although a file organization of relative is specified, RMS would in fact obtain the file organization from an existing file. If the file organization were not relative, the file open statement would fail.
- 3. The call to the FIND routine positions the file at the component specified in REC\_NUM. ERROR := CONTINUE is specified, because the program will attempt to handle errors from the FIND procedure.

4. The STATUS function is used to detect file access errors. The program performs certain operations if the status is SUCCESS or ERRDURFIN. If any other errors occur, LIB\$STOP is called.

### **Source Program Number 3:**

```
PROGRAM Indexed (INPUT, OUTPUT);
     INDEXED.PAS
                                                        *)
    This program illustrates keyed and sequential
                                                         *)
    access to an indexed sequential file. It prompts
                                                        *)
    for a department number then prints the name of
                                                         *)
     each person in the department.
                                                        *)
LABEL 100:
CONST Prompt = 'Enter department no. (119,210,220): ';
TYPE Employee_Rec = RECORD
                     Id_Num: PACKED ARRAY [1..5] OF CHAR;
                     Name: PACKED ARRAY [1..6] OF CHAR;
                     Dept: PACKED ARRAY [1..3] OF CHAR;
                     Skill: PACKED ARRAY [1..2] OF CHAR;
                     Salary: PACKED ARRAY [1..4] OF CHAR;
                     END;
VAR Employees : FILE OF Employee_Rec;
     Dept_Key : PACKED ARRAY [1..3] OF CHAR;
BEGIN
   (* Open indexed file *)
   OPEN (Employees, 'IDX.DAT', History:= OLD,
         Organization:= INDEXED, Access_Method:= KEYED);
   (* Obtain department number *)
   WRITELN ('Type CTRL/Z to quit');
   WRITE (Prompt);
   WHILE NOT EOF (INPUT) DO
     BEGIN
         READLN (Dept_key);
         (* Read first dept_key record *)
                                                             2
         FINDK (Employees, Key_Number:= 0,
               Key_Value:= Dept_Key);
```

```
(* If valid dept., output members *)
         IF NOT UFB (Employees)
            THEN (* valid dept number *)
              WHILE NOT EOF (Employees) DO
                     IF Employees . Dept <> Dept_Key
                     THEN GOTO 100;
                                                              8
                  WRITELN (Employees . Name);
                  GET (Employees)
                       (* output dept. members *)
            ELSE (* not valid dept. number *)
               WRITELN ('Invalid department number.');
100:
         (* Prompt for another dept. number *)
         WRITE (Prompt)
      END (* not CTRL/Z *)
END.
Sample Use:
$ RUN INDEXED
Type CTRL/Z to quit
Enter department no. (119,210,220): 119
ANDEWF
DALLJE
FLANJE
FLINGA
GREEJW
JONEKB
MANKCA
MARSJJ
REDFBB
SCHAWE
WIENSH
Enter department no. (119,210,220): 3
Invalid department number.
Enter department no. (119,210,220): ~Z
```

- To allow random keyed access to an indexed file, the ACCESS\_ METHOD parameter to the OPEN procedure must be INDEXED.
  - The location of the keys in the records need not be specified since this information is obtained by RMS from the file prologue.
- 2. The FINDK procedure is used to locate a record by key value. A key\_ number of zero indicates the primary key, and key\_value specifies the value of the key.

- It is not necessary to call RESETK, because the file need not be in inspection mode before FINDK is executed.
- Sequential reads are performed to obtain the records with the appropriate department number. VAX RMS will not return a different status code when the key value changes. Consequently, the program must check the key value in the record to determine whether or not it has changed.

# F.8 Measuring and Improving Performance

This example illustrates how to adjust the size of the process working set from a program.

### **Source Program:**

```
[INHERIT ('SYS$LIBRARY:STARLET.PEN')] PROGRAM Adjust( OUTPUT);
     ADJUST.PAS
     This program illustrates how a program can control its
                                                              *)
                                                              *)
     working set size using the $ADJWSL system service.
CONST
        Index = 10;
VAR
        Adjust_Amt:
                        INTEGER:
        New_Limit:
                        INTEGER;
        Sys_Stat:
                        INTEGER:
PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER); EXTERN;
BEGIN
   Adjust_Amt: =-50;
   WHILE Adjust_Amt <= 70 DO
         (* Modify working set limit *)
                                                              0
         Sys_Stat:= $ADJWSL(Adjust_Amt, New_Limit);
         IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
         WRITELN ('Modify working set by ', Adjust_Amt,
            ' New working set size = ', New_Limit);
         Adjust_Amt := Adjust_Amt + 10
      END
END.
```

### Sample Use:

```
$ SHOW WORKING SET
  Working Set
                /Limit= 150
                               /Quota= 200
                                                       /Extent=
                                                                200
  Adjustment enabled Authorized Quota= 200 Authorized Extent=
$ RUN ADJUST
Modify working set by -50
                            New working set size = 100
Modify working set by -40
                            New working set size = 60
Modify working set by -30
                            New working set size = 40
Modify working set by -20
                                                           8
                            New working set size = 40
Modify working set by -10
                            New working set size = 40
Modify working set by 0
                            New working set size = 40
Modify working set by 10
                            New working set size = 50
Modify working set by 20
                            New working set size = 70
Modify working set by 30
                            New working set size = 100
Modify working set by 40
                            New working set size = 140
Modify working set by 50
                            New working set size = 190
Modify working set by 60
                            New working set size = 200
Modify working set by 70
                           New working set size = 200
```

#### **Notes:**

- 1. The \$ADJWSL is used to increase or decrease the number of pages in the process working set.
- 2. The SHOW WORKING\_SET command in DCL displays the current working set limit and the maximum quota.
- 3. Notice that the program cannot decrease the working set limit beneath the minimum established by the operating system.
- 4. Similarly, the process working set cannot be expanded beyond the authorized quota.

# F.9 Accessing Help Libraries

The following example illustrates how to obtain text from a help library. After the initial help request has been satisfied, the user is prompted and can request additional information.

#### **Source Program:**

```
PROGRAM Gethelp (INPUT, OUTPUT);
(*
        GETHELP. PAS
(*
        This program will look up help text in
        SYS$HELPLIB.HLB and displays it.
(*
CONST LBR$C_READ = "X'01';
TYPE Word_Integer = [WORD] 1..65535;
     Char_String = PACKED ARRAY [1..32] OF CHAR;
     Int_Array = ARRAY [1..3] OF INTEGER;
             : ARRAY [1..3] OF Char_String;
     Key_Len : ARRAY [1..3] OF INTEGER;
     Lib_Index, Help_Stat, Counter: INTEGER;
(* Declare external RTL routines *)
FUNCTION LBR$INI_CONTROL (VAR Library_Index : INTEGER;
                          Func : INTEGER;
                          Libe_Type : INTEGER := %IMMED O;
                          VAR Namblk: ARRAY [1..U:INTEGER]
                                OF INTEGER := %IMMED O):
                          INTEGER; EXTERN;
FUNCTION LBR$OPEN (Library_Index : INTEGER;
                   Fns : [CLASS_S] PACKED ARRAY [1..U:INTEGER]
                         OF CHAR := %IMMED O;
                   Create_Options: Int_Array := %IMMED 0;
                   Dns : [CLASS_S] PACKED ARRAY [12..U2:INTEGER]
                         OF CHAR := %IMMED O;
                   Rlfna : ARRAY [13..U3:INTEGER] OF INTEGER
                         := %IMMED 0:
                   Rns : [CLASS_S] PACKED ARRAY [14..U4:INTEGER]
                         OF CHAR := %IMMED O;
                   VAR Rnslen : INTEGER := %IMMED 0): INTEGER;
                                EXTERN:
FUNCTION LBR$GET_HELP (Library_Index : INTEGER;
                       Line_Width : INTEGER := %IMMED O;
                       %IMMED [UNBOUND] PROCEDURE Routine
                              := %IMMED 0;
                       data : INTEGER := %IMMED O;
                       Key_1 : [CLASS_S] PACKED ARRAY [1..U:
                               INTEGER] OF CHAR;
                       Key_2 : [CLASS_S] PACKED ARRAY [12..U2:
                               INTEGER] OF CHAR;
                       Key_3 : [CLASS_S] PACKED ARRAY [13..U3:
                               INTEGER] OF CHAR): INTEGER;
                               EXTERN;
```

```
FUNCTION LBR$CLOSE (Library_Index : INTEGER): INTEGER; EXTERN;
PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER); EXTERN;
BEGIN
                (* main program *)
   (* Initialize the librarian *)
   Help_Stat := LBR$INI_CONTROL (Lib_Index, LBR$C_READ,,);
   IF NOT ODD (Help_Stat) THEN LIB$STOP (Help_Stat);
   (* Open the library *)
   Help_Stat := LBR$OPEN (Library_Index := Lib_Index,
                          Fns := 'SYS$HELP:HELPLIB.HLB');
   IF NOT ODD (Help_Stat) THEN LIB$STOP (Help_Stat);
   (* Get HELP keys *)
   FOR Counter := 1 TO 3 DO
      BEGIN
      WRITE ('Enter key: ');
                                                              4
      READLN (Key[Counter]);
   (* Locate and print the help text *)
   Help_Stat:=LBR$GET_HELP (Lib_Index,,,,Key[1], Key[2], Key[3]);
   IF NOT ODD (Help_Stat) THEN LIB$STOP (Help_Stat);
END.
```

#### Sample Use:

```
$ RUN GETHELP
Enter Key : REQUEST
Enter Key : /REPLY
Enter Key :
 REQUEST
     /REPLY
     Requests a reply to the specified message.
     If you request a reply, the message is assigned a unique
      identification so that the operator can respond. You will
     not be able to continue until the operator responds, unless
     you use CTRL/Y.
```

#### **Notes:**

1. The LBR\$C\_READ symbolic code is defined as a constant (with a hexidecimal value). The value associated with this particular code is obtained from the \$LBRDEF macro in STARLET.MLB.

The code is used to specify the operation that is to be performed on a library.

- 2. The call to LBR\$INL\_CONTROL inititalizes the library index and defines the operation to be performed on the library.
- 3. The call to LBR\$OPEN identifies the library to be accessed, in this case the system help library, SYS\$HELP:HELPLIB.HLB.
- 4. The user is asked to supply the key(s) to look up help text. In this case Key[1] corresponds to a DCL command, while Key[2] and Key[3] (optional) are command parameters or qualifiers.
- If no routine name is specified in the call to LBR\$GET\_HELP, the returned help text is written to SYS\$OUTPUT (with line-width defaulting to 80). Note that several special symbols can be used for the key arguments:
  - All first-level help text in the library.
  - KEY... All help text with specified key and its subkeys.
    - All help text in the library.

#### F.10 **Creating and Managing Other Processes**

The following example illustrates the ability of a created process to use the SYS\$GETJPIW system service to obtain the PID of its creator process.

#### Source Program Number 1:

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Getjpi( OUTPUT);
    GETJPI.PAS
(*
    This program illustrates process creation and control.
                                                            *)
(* It creates a subprocess then hibernates until the
                                                            *)
    subprocess wakes it.
                                                            *)
       File_Name = 'GETJPISUB';
       Sub_Name = 'OSCAR';
       Esc_Null = ''(27)''(0)''; (* ESCAPE/NULL string *)
```

```
TYPE $UBYTE = [BYTE] 0..255;
       Word_Integer = [WORD] 0..65535;
       Dummy_Rec = RECORD
       Item_List = RECORD
                   Buf_Len1
                                : Word_Integer;
                    Item_Code1 : Word_Integer;
                    Buffer_Addr1 : [UNSAFE] ^[UNSAFE]Dummy_Rec;
                    Ret_Len_Addr1: ^Word_Integer;
                    Buf_Len2 : Word_Integer;
                    Item_Code2 : Word_Integer;
                    Buffer_Addr2 : ^INTEGER;
                   Ret_Len_Addr2: ^Word_Integer;
                    Terminator : INTEGER
                    END;
       Terminal : [VOLATILE] VARYING [255] OF CHAR;
VAR
       Equiv_Name: [VOLATILE] PACKED ARRAY [1..255] OF CHAR;
        Process_Id: UNSIGNED;
        New_Term_Len: Word_Integer;
        Equiv_Name_Len, Info_Len : [VOLATILE] Word_Integer;
        Tran_Mask :
                    [VOLATILE] INTEGER;
        Sys_Stat : INTEGER;
        Not_Terminal: BOOLEAN := TRUE;
        Trn_List : Item_List;
        Unsigned_Mask, Term_Attr : UNSIGNED;
PROCEDURE LIB$STOP (%IMMED Cond_Value: INTEGER); EXTERN;
BEGIN
   (* Initialize *)
   Term_Attr:=LNM$M_TERMINAL;
   Process_Id:= 0;
   Terminal: = 'SYS$OUTPUT';
   Trn_List.Buf_Len1 := 255;
   Trn_List.Item_Code1 := LNM$_STRING;
Trn_List.Buffer_Addr1 := ADDRESS (Equiv_Name);
   Trn_List.Ret_Len_Addr1 := ADDRESS (Equiv_Name_Len);
   Trn_List.Buf_Len2 := 4;
   Trn_List.Item_Code2 := LNM$_ATTRIBUTES;
   Trn_List.Buffer_Addr2 := ADDRESS (Tran_Mask);
   Trn_List.Ret_Len_Addr2 := ADDRESS (Info_Len);
   Trn_List.Terminator := 0;
```

```
(* Translate SYS$OUTPUT *)
   WHILE Not_Terminal AND (Sys_Stat <> SS$_NOLOGNAM) DO
       BEGIN
       Sys_Stat:= $TRNLNM (Tabnam := 'LNM$PROCESS'.
                           Lognam := Terminal,
                           Itmlst := Trn_List);
       Terminal := SUBSTR (Equiv_Name,1,Equiv_Name_Len);
       Unsigned_Mask := Tran_Mask;
       IF UAND (Unsigned_Mask,Term_Attr) <> 0
           THEN Not_Terminal := FALSE:
       END:
   (* Check if process permanent file *)
   IF (SUBSTR (Terminal,1,2) = Esc_Null)
      THEN
         BEGIN
         (* Strip off extra 4 bytes *)
         New_Term_Len:= Equiv_Name_Len - 4;
         Terminal:= SUBSTR (Terminal, 5, New_Term_Len);
         END:
   (* Create the subprocess *)
   Sys_Stat:=$CREPRC (Pidadr := Process_Id, Image:=File_Name, 1
                      Input := Terminal, Output:= Terminal,
                      Prcnam := Sub_Name, Baspri:= 4);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   WRITELN ('PID of subprocess OSCAR is ', HEX(Process_Id));
   (* Wait for wakeup by subprocess *)
   Sys_Stat:= $HIBER;
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   WRITELN ('GETJPI has been awakened.')
END.
Source Program Number 2:
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Getjpisub( OUTPUT);
    GETJPISUB. PAS
                                                            *)
(* This program is run in the subprocess OSCAR which is
                                                            *)
(*
    created by GETJPI. It obtains its creator's PID then
                                                            *)
    wakes it.
                                                            *)
```

```
Word_Integer = [WORD] 0..65535;
TYPE
       Getjpi_List = RECORD
                      Buf_Len
                                 : Word_Integer;
                      Item_Code : Word_Integer;
                      Buffer_Addr : ^UNSIGNED;
                      Ret_Len_Addr: ^INTEGER;
                      Must_Be_Zero: INTEGER;
                      END;
        Io_Block = RECORD
                   Io_Stat, Count: Word_Integer;
                   Dev_Info : INTEGER;
                   END:
        Jpi_List: Getjpi_List;
VAR
        Stat_Block: Io_Block;
        Sys_Stat: INTEGER;
        Buf_Val : [VOLATILE] UNSIGNED;
        Ret_VAl : [VOLATILE] INTEGER;
PROCEDURE LIB$STOP (%IMMED Cond_Value: INTEGER); EXTERN;
   (* Initialize *)
   Jpi_List.Buf_Len
                        := 4:
   Jpi_List.Item_Code := JPI$_OWNER;
   Jpi_List.Buffer_Addr := ADDRESS (Buf_Val);
   Jpi_List.Ret_Len_Addr:= ADDRESS (Ret_Val);
   Jpi_List.Must_Be_Zero:= 0;
   (* Get PID of creator *)
   Sys_Stat:= $GETJPI (Efn:= 1, Iosb:=Stat_Block, Itmlst:=
                       Jpi_List);
                                                             8
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Sys_Stat:= $SYNCH (Efn:=1, Iosb:=Stat_Block);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   IF NOT ODD (Stat_Block.Io_Stat) THEN
              LIB$STOP (Stat_Block.Io_Stat);
   (* Wake creator *)
                                                             4
   WRITELN (' OSCAR is waking creator.');
   Sys_Stat:= $WAKE (Jpi_List.Buffer_Addr^,);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat)
END.
Sample Use:
$ RUN GETJPI
PID of subprocess OSCAR is OOO2002E
   OSCAR is waking creator.
GETJPI has been awakened.
```

#### Notes:

- 1. The subprocess is created using SYS\$CREPRC.
- 2. The item list for \$GETJPI consists of a single item descriptor followed by a zero longword. JPI\$\_OWNER is the item code which requests the PID of the owner process. If there is no owner process (i.e., if the process about which information is requested is a detached process), the system service \$GETJPI returns a PID of zero.
- 3. Because the item code JPI\$\_OWNER is in the item list, \$GETJPI returns the PID of the owner of the process about which information is requested. If the item code were IPI\$\_PID, \$GETIPI would return the PID of the process about which information is requested.
  - Because the default value of 0 is used for arguments PIDADR and PRCNAM, the process about which information is requested is the requesting process, namely, OSCAR.
- 4. OSCAR wakes its owner process using the PID it received from the call to \$GETIPI.

#### F.11 **Translating Logical Names**

The following example illustrates the common task of translating logical names.

#### **Source Program**

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Translate(INPUT, OUTPUT);
VAR
   Input_String : VARYING [132] OF CHAR;
  Output_String : VARYING [132] OF CHAR;
  Return_Status : INTEGER;
  Trnlnm_Item_List : RECORD
                      Buffer_Length : [WORD] 0..65535;
                      Item_Code : [WORD] 0..65535;
                      Buffer_Address: INTEGER;
                      Return_Length : INTEGER;
                      End_Of_List : INTEGER;
                      END:
BEGIN
(* Prompt user for logical name to translate *)
WRITE('Logical name? ');
READLN(Input_String);
(* Build item list *)
WITH Trnlnm_Item_List DO
   BEGIN
   Buffer_Length := SIZE(Output_String.Body);
               := LNM$_STRING;
   Item_Code
   Buffer_Address:= IADDRESS(Output_String.Body);
   Return_Length := IADDRESS(Output_String.Length);
    End_Of_List := 0;
    END;
(* Call TRNLNM to translate logical name *)
RETURN_STATUS := $TRNLNM(
            := LNM$M_CASE_BLIND,
     ATTR
     TABNAM := 'LNM$SYSTEM_TABLE',
     LOGNAM := Input_String,
      ITMLST := Trnlnm_Item_List);
IF RETURN_STATUS = SS$_NORMAL
THEN
      WRITELN('Translated name is ',Output_String)
ELSE IF RETURN_STATUS = SS$_NOLOGNAM
THEN
      WRITELN('No translation for logical name ', Input_String)
ELSE
      WRITELN('Error in TRNLNM routine');
END.
```

#### Sample Use:

**\$ RUN TRNLNM** 

Logical name? sys\$batch Translated name is CLU\_BATCH

\$ RUN TRNLNM

Logical name? hello No translation for logical name hello

#### **Notes:**

1. Build the item list to place the result directly into a VARYING OF CHAR variable. This is done by computing the address of the length word and body.

# **INDEX**

A	ARRAY type allocation size of ● 1-11
	packing • 1-17
Access method	AST routines
direct • 4-5	attributes required for • 3-7
file • 4-2	ASYNCHRONOUS attribute
keyed ● 4-24	required for condition handler • 5-6
sequential • 4-12, 4-24	use with Run-Time Library routines •
Access-method parameter	3-32
of OPEN procedure • 4-12	Asynchronous system traps • 3-7
Actual parameter	
definition of • 3-8	В
mechanism specifier on • 3-20	
ADDRESS function	BITNEXT function ● 1-14
conflict with optimization • 2-14	BITSIZE function • 1-14
Addresses	Block
as foreign mechanism parameters • 3-22	allocation of • 1-7
required by PASCAL routine • 3-22	BOOLEAN type
ALIGNED attribute • 1-16	allocation size of ● 1-11
Alignment	Bound procedure value • 3-13
of variables • 1-16	required by PASCAL routine • 3-22
Alignment attributes • 1-16	Branch
Allocation	to longword-aligned address • 2-10
attributes ● 1-6	By-descriptor mechanism • 3-14
in common block • 1-4	By-immediate-value mechanism
in program section ● 1-2	%IMMED specifier with ● 3-13
of subrange component • E-9	IMMEDIATE attribute with ● 3-14
of variables • 1-6, 1-10	By-reference mechanism • 3-9
sizes • 1-10	%REF specifier with ● 3-12
Alternate keys   4-24	REFERENCE attribute with • 3-13
characteristics of ● 4-25	value semantics with • 3-10
Argument pointers	variable semantics with • 3-11
saved by routine call • 3-3	By-string-descriptor mechanism • 3-18,
Array	3-19
multidimensional, packing • E-8	

# C

Call frame • 3-3 Call stack • 3-3 CALLG instruction • 3-4 Calling VAX/VMS system services • 3-23 Calling Run-Time Library routines • 3-32 CALLS instruction • 3-4 Carriage-control parameter of OPEN procedure • 4-13 CASE statement use for efficiency • 2-11 CHAR type allocation size of • 1-11 Character strings as function results • 3-2 reading • E-10 CHECK attribute avoid for efficiency • 2-12 CLOSE procedure • 4-20 CLOSE procedure parameters disposition • 4-20 error-recovery • 4-21 user-action • 4-20 \$CODE program section • 1-2, 1-7 Comments equivalence of delimiters • E-8 COMMON attribute • 1-4 Common block • 1-4 properties of • 1-5 Communication interprocess • 4-36 remote • 4-37 task-to-task • 4-37 Compile-time qualifier • E-6 Compiler diagnostics • A-1 generated labels • 2-10 Component numbers in relative files • 4-3 Condition handlers by Run-Time Library • 5-2 controlling execution • 5-5 established by VAX/VMS • 5-3 establishing • 5-6 examples of • 5-12

Condition handlers (cont'd.) for faults • 5-11 for traps • 5-11 parameters to • 5-7 performing I/O from • 5-6 removing • 5-7 reporting conditions • 5-5 return value of • 5-9 Condition signal • 5-4 Condition symbol definition of • 3-34 Condition value • 3-27, 5-10 contained in file • 3-34 for faults • 5-12 for traps • 5-12 matching • 5-11 severity code of • 5-10 Constants allocation of • 1-7 compile-time evaluation of • 2-3 use for efficiency • 2-11 Conversion of constants • 2-3 \$CREATE/FDL utility use to create indexed file • 4-24 Current component pointer • 4-26

#### D

D\_floating double-precision representation of • 1-21 Data structure parameter • 3-25 Debugging effects of optimization • 2-16 DECnet network • 4-37 Decommitted features • E-2 Default file name of OPEN procedure • 4-19 Default parameter • 3-27 Default size for text files • 4-11 Delayed device access • 4-34 DELETE procedure • 4-30 %DESCR mechanism specifier • 3-19 on actual parameter variable • 3-21 Descriptor mechanism • 3-14

Descriptors	EXTERNAL attribute		
%DESCR mechanism specifier ● 3-14,	effect on routine call • 3-7		
3-19	similarity to UNBOUND • 3-7		
on actual parameters • 3-21	External routines		
required by PASCAL routine • 3-22	calling PASCAL routine from • 3-22		
Direct access to files • 4-5	passing parameters to • 3-7		
Disposition parameter	F		
of CLOSE procedure • 4-20			
of OPEN procedure • 4-15			
DOUBLE type	FAB		
allocation size of ● 1-11	access to • D-1		
representation of ● 1-21	Fault		
Double-precision data	condition handling for • 5-11		
representation of ● 1-21	convert to trap • 5-12		
Duplicate keys    4-27	Field width		
Dynamic array • E-2	default in previous version • E-11		
see also conformant array	File		
compatibility of • E-3	passing as actual parameter • 3-11		
contrasted with conformant array • E-3	File access block		
use of LOWER with • E-3	see FAB		
use of UPPER with • E-3	File components		
	current ● 4-26		
E	deleting • 4-30		
	lengths ● 4-11		
End-of-line	locked • 4-23		
complete record with • 4-32, 4-33	next ● 4-26		
while reading string • E-10	File name parameter		
Enumerated type	of OPEN procedure • 4-11		
allocation size of ● 1-11	FILE OF CHAR		
ENVIRONMENT attribute	contrasted to TEXT • 4-33		
effect on allocation ● 1-6	FILE type		
Error messages	allocation size of ● 1-11		
compiler ● A-1	File variable parameter		
run-time • A-56	of OPEN procedure • 4-10		
ERROR parameter • 4-19	Files		
of CLOSE procedure • 4-21	access methods • 4-12		
Error-recovery parameter	carriage control parameter • 4-13		
in CLOSE procedure • 4-21	components of • 4-3		
of OPEN procedure • 4-19	current component of • 4-26		
ESTABLISH procedure • 5-6	disposition of • 4-20		
Exception conditions	disposition parameter • 4-15		
on indexed files • 4-31	filling buffer of ● 4-34		
Expressions	history parameter • 4-11		
	length of components in • 4-11, 4-13		
optimization of • 2-8, 2-13	locked component in • 4-23		
order of evaluation ● 2-8	next component of • 4-26		
reordering of ● 2-2	organization of • 4-2, 4-14		
	Organization of the transfer		

Files (cont'd.) Н passed to PASCAL routine • 3-22 reference to • 4-34 sharing • 4-15, 4-22 History parameter FIND procedure of OPEN procedure • 4-11 for direct access • 4-5, 4-12 FINDK procedure for keyed access • 4-5, 4-12 use on indexed file • 4-28 IADDRESS function Fixed-length record • 4-6 conflict with optimization • 2-14 Floating-point data IF-THEN-ELSE statement representation of • 1-19 use for efficiency • 2-11 FOR statement %IMMED mechanism specifier • 3-13 use for efficiency • 2-11 on actual parameter variable • 3-21 Foreign mechanism specifier • 3-8 on default parameter • 3-27 Foreign semantics • 3-12 on routines • 3-14 implied by %DESCR • 3-19 IMMEDIATE attribute • 3-14 implied by %REF • 3-12 Immediate value mechanism • 3-13, 3-14 implied by %STDESCR • 3-18 %INCLUDE directive Formal parameter default file type for • E-8 definition of • 3-8 Indexed files mechanism specifier on • 3-8 creating with \$CREATE/FDL utility • Frame pointer of UNBOUND routine • 3-7 creating with OPEN procedure • 4-24 saved by routine call • 3-3 deleting components from • 4-30 **Functions** exception conditions with • 4-31 methods of returning result • 3-2 key field in • 4-3 optimization of • 2-7 keved access to • 4-24 passed as actual parameter • 3-13 locked component in • 4-23 passed by immediate value • 3-14 organization • 4-3, 4-14, 4-24 user-action reading from • 4-28 to close a file • 4-20 RMS pointers in • 4-26 to open a file • 4-16 sequential access to • 4-4, 4-24 updating • 4-29 G writing to • 4-27 **INHERIT** attribute G\_floating double-precision representation • use to obtain system definitions • 3-30 1-22 **INITIALIZE** attribute **GET** procedure effect on allocation • 1-7 for sequential access • 4-12 effect on routine call • 3-7 GLOBAL attribute similarity to UNBOUND • 3-7 effect on routine call • 3-7 Input procedures similarity to UNBOUND • 3-7 ERROR parameter with • 4-19 Global identifier Integer overflow in previous version • E-11 checking of • 2-12 GOTO statement INTEGER type avoid for efficiency • 2-11 allocation size of ● 1-11

Interprocess communication • 4-36

K

Mechanis
Mechanis
Mechanis
%DES

KEY attribute • 4-24
Key field
alternate • 4-24
characteristics of • 4-25
defining • 4-24
duplicating • 4-27
in indexed file • 4-3
primary • 4-3, 4-24

Keyed access method • 4-5, 4-24

Lazy lookahead • 4-34
LIB\$ESTABLISH routine
use to establish condition handler • 5-6
LIB\$MATCH\_COND function • 5-11

LIB\$SIGNAL procedure use to signal condition • 5-4 LIB\$SIM\_TRAP procedure • 5-12 LIB\$STOP procedure use to signal condition • 5-4 LIBDEF.PAS definition file • 3-34 Line buffer • 4-32 LIST attribute use with Run-Time Library routine • 3-33 \$LOCAL program section • 1-2 LOCATE procedure for direct access • 4-5, 4-12 Locked record • 4-23 in indexed file • 4-23 in relative file • 4-23 unlocking • 4-23 Logical expressions optimization of • 2-8 Logical names equated with local file • 5-38 equated with remote file • 5-38 LOWER function • E-3



Mailboxes • 4-36 MAXINT • 2-14

Mechanism arrays • 5-7 Mechanism specifier • 3-9 Mechanism specifiers %DESCR • 3-19 %IMMED • 3-13 IMMEDIATE • 3-14 on actual parameters • 3-20 on formal parameters • 3-8 %REF • 3-12 REFERENCE • 3-13 %STDESCR • 3-18 Mechanisms by-descriptor • 3-14 by-immediate-value %IMMED specifier • 3-13 IMMEDIATE attribute • 3-14 by-reference • 3-9 %REF specifier with • 3-12 REFERENCE attribute • 3-13 value semantics with • 3-10 variable semantics with • 3-11 by-string-descriptor • 3-18, 3-19 MOD operator decommitted definition of • E-10 MTHDEF.PAS definition file • 3-34 Multidimensional array effect of packing • E-8 packing • 1-17

#### N

Network • 4-37

Next component pointer • 4-26

NEXT function • 1-14

/NOOPTIMIZE attribute

use of in optimization • 2-13

NOOPTIMIZE attribute

use of in optimization • 2-13

## 0

/OLD\_VERSION qualifier • E-7
OPEN procedure • 4-8
decommitted syntax of • E-5
use to create indexed file • 4-24
OPEN procedure parameters
access-method • 4-12

OPEN procedure parameters (cont'd.)	Parameters			
carriage-control • 4-13	by-descriptor mechanism • 3-14			
default file name • 4-19	by-immediate-value mechanism • 3-13			
disposition ● 4-15				
error-recovery • 4-19	by-reference mechanism • 3-12, 3-13			
file name • 4-11	data structure • 3-25			
file sharing • 4-15	default value for • 3-27			
file variable • 4-10	definition of actual • 3-8			
history • 4-11	definition of formal • 3-8			
organization ● 4-14	of OPEN procedure • 4-9			
record-length • 4-11	passing from non-PASCAL routines • 3-22 passing to external routines • 3-7			
record-type • 4-13				
user-action • 4-16				
Operations	passing to Run-Time Library routines			
optimization of • 2-8	3-33			
type cast • 2-16	passing to system services • 3-24			
Optimization • 2-1	Parentheses			
considerations • 2-13	use for efficiency • 2-11			
definition of • 2-1	PAS\$FAB function • D-1			
disabling during recompilation • E-7	PAS\$GLOBAL program section • 1-2 PAS\$MARK function • D-3 PAS\$RAB function • D-2			
disabling for efficiency • 2-13				
effect on debugging • 2-16				
kinds of • 2-2	PAS\$RELEASE function ● E-4 Pointer type			
reducing errors through • 2-10				
Organization of files • 4-2	allocation size of ● 1-11			
indexed • 4-3, 4-14	POS attribute			
relative • 4-3, 4-14	with data structure parameter • 3-25			
sequential • 4-2, 4-14	Predeclared functions			
Organization parameter	optimization of • 2-7			
of OPEN procedure • 4-14	Primary keys			
Output procedures	characteristics of ● 4-25			
ERROR parameter with • 4-19	in indexed file • 4-3			
Overflow checking	Procedure			
avoid for efficiency • 2-12	calling standard • 3-1			
OVERLAID attribute	passed as actual parameter • 3-13			
effect on allocation ● 1-6	passed by immediate value • 3-14			
P	Procedure parameter notation • C-1			
F	Program counter			
DACKED ADDAY OF CHARA	saved by routine call • 3-3			
PACKED ARRAY OF CHAR type	Program sections			
as function result type • 3-2	allocation in • E-11			
Packing	default ● 1-2			
multidimensional arrays • 1-17	properties of ● 1-4			
of structured types • 1-17	use of ● 1-1			
Parameter	Programming			
dynamic array ● E-2	considerations ● 2-11			
to condition handler • 5-7				
Parameter list • 3-2				

#### 6-Index

Prompting	Records				
of text file • 4-33	in RMS • 4-32				
Propagation	variant • 2-15				
value ● 2-9	Records in files fixed-length • 4-6, 4-13				
PSECT attribute ● 1-3					
PUT procedure	stream • 4-13				
for sequential access • 4-12	variable-length • 4-6, 4-13				
on indexed file • 4-27	%REF mechanism specifier • 3-12				
	on actual parameter variable • 3-21				
Q	REFERENCE attribute • 3-13				
_	Reference mechanism • 3-9				
Quadruple-precision data	Registers				
representation of • 1-11, 1-23	assignment of variables to • 2-2				
Qualifiers	contents saved by routine call • 3-3				
compile-time • E-6	effects of optimization • 2-16				
in source code • E-6	Relative files • 4-3				
/OLD_VERSION ● E-7	component number in • 4-3				
/ OLD_VENSION - E-/	direct access to • 4-5				
R	locked component in • 4-23				
••	Remote communication • 4-37				
DAD	REPEAT statement				
RAB	use for efficiency • 2-11				
access to • D-2	RESET procedure				
READ procedure	for sequential access • 4-12				
for sequential access • 4-12	RESETK procedure				
with character string • E-10	for keyed access • 4-5, 4-12				
READONLY attribute	use on indexed file • 4-28				
effect on allocation • 1-7	Resource sharing • 4-37				
on pointer variables • 2-14	Return values				
with %DESCR parameter • 3-19	function • 3-2				
with %REF parameter • 3-12	Run-Time Library routines • 3-32				
with %STDESCR parameter • 3-18	REVERT procedure • 5-7				
with REFERENCE parameter • 3-13	REWRITE procedure				
with system services • 3-24	for sequential access • 4-12				
with variable semantics • 3-11	Routines				
REAL type	(AST) • 3-7				
allocation size of • 1-11	external				
representation of • 1-19	calling PASCAL routine from • 3-22				
Record access • 4-2	passing parameters to • 3-7				
Record access block	instructions for calling • 3-4				
see RAB	passed as actual parameter • 3-13				
Record locking • 4-23	passed as actual parameter 3-13				
RECORD type	Run-time				
allocation size of ● 1-11	diagnostics • A-56				
packing • 1-17	Run-time error				
Record-length parameter	reporting of • 5-2				
of OPEN procedure ● 4-11	Run-Time Library				
	calling routines from • 3-32				

Run-Time Library (cont'd.)	Size attributes			
condition symbols • 3-34	effect on allocation • 1-11			
declaring in PASCAL • C-1 parameters to routine from • 3-32	with data structure parameter • 3-25 SIZE function • 1-14			
				symbol definitions for • 3-34
using LIST attribute with • 3-33	returned by condition handler • 5-9			
0	SS\$_RESIGNAL return value			
S	returned by condition handler • 5-9			
	_ STARLET files			
Semantics	contents of ● 3-30			
foreign • 3-12	Statements			
implied by %DESCR • 3-19	optimization of ● 2-6			
implied by %REF • 3-12	STATIC attribute			
implied by %STDESCR • 3-18	effect on allocation ● 1-7			
value ● 3-10	STATUS function			
with system services • 3-24	conditions detected by • B-1			
variable ● 3-11	testing indexed file with • 4-31			
implied by %DESCR • 3-19	use after READLN • 4-7			
implied by %REF • 3-12	%STDESCR mechanism specifier • 3-18			
implied by %STDESCR • 3-18	on actual parameter variable • 3-21			
implied by REFERENCE • 3-13	Storage Allocation • 1-6, 1-10			
with system services • 3-24	Stream record format • 4-7			
Sequential access • 4-2	STREAM_CR ● 4-7			
methods • 4-24	STREAM_LF ● 4-7			
to indexed file • 4-5	String-descriptor mechanism • 3-18, 3-19			
to relative file • 4-5	Structured statements			
Sequential files	optimization of • 2-6			
direct access to • 4-5	use for efficiency • 2-11			
organization parameter • 4-14	Structured type			
SET type	as function result type • 3-2			
allocation size of ● 1-11	packing of ● 1-17			
storage of when unpacked • E-9	Subexpressions			
Sharing	optimization of ● 2-4			
of files • 4-22	Subrange type			
of resources • 4-37	allocation size of ● 1-11			
Sharing parameter	Symbol definition files • 3-34			
of OPEN procedure • 4-15	Symbolic constants			
SIGDEF.PAS definition file • 3-34	allocation of ● 1-7			
Signal arrays • 5-7	SYS\$UNWIND function			
in condition handlers • 5-7	called by condition handler • 5-9			
SINGLE type	System services			
allocation size of ● 1-11	ASYNCHRONOUS required • 3-23			
representation of • 1-19	called as procedure • 3-27			
Single-precision data	calling from PASCAL • 3-23			
representation of • 1-19	condition value of • 3-27			
Size	data structure parameter • 3-25			
default values for text files • 4-11	definition file for • 3-30			
default values for text files 4-11	optional parameters for • 3-27			

System services (cont'd.)
parameters to • 3-23
passing parameters to • 3-24
VAX/VMS • 3-23
System symbol definitions • 3-34

#### T

Temporary variables avoid for efficiency • 2-12 **Terminal** output to • 4-33 Text file buffering of characters in • 4-32 component of • 4-32 contrasted with FILE OF CHAR • E-10 delayed device access to • 4-34 input and output on • 4-32 output to terminal • 4-33 prompting of • 4-33 use to supply input • 4-34 TEXT type 32-bit rule • 1-10 Trap condition handling for • 5-11 TRUNCATE procedure for sequential access • 4-12 Type cast operation optimization of • 2-16 Type conversion of constants • 2-3

## īī

UFB function
on indexed file • 4-29
testing indexed file with • 4-31
UNALIGNED attribute • 1-16
UNBOUND attribute
effect on routine call • 3-7
UNLOCK procedure • 4-23
UNSIGNED type
allocation size of • 1-11
Unwind
of stack by condition handler • 5-9
UPDATE procedure
on indexed file • 4-29

UPPER function ● E-3
User-action function
parameters to ● 4-16, 4-21
to close a file ● 4-20
to open a file ● 4-16
use to create indexed file ● 4-24
User-action parameter
of CLOSE procedure ● 4-20
of OPEN procedure ● 4-16

#### V

Value semantics • 3-10 with system services • 3-24 Variable semantics • 3-11 implied by %DESCR • 3-19 implied by %REF • 3-12 implied by %STDESCR • 3-18 implied by REFERENCE • 3-13 with system services • 3-24 Variable-length record • 4-6 Variables alignment of • 1-16 allocation in common block • 1-4, 1-6 allocation in program section • 1-6 allocation of ● 1-10 avoid for efficiency • 2-12 pointer • 2-14 storage allocation • 1-6 VARYING OF CHAR type allocation size of ● 1-11 as function result type • 3-2 representation of • 1-18 VAX Procedure Calling Standard • 3-1 VAX Run-Time Library calling routines from • 3-32 VAX/VMS system service calling from PASCAL • 3-23 Visibility attributes effect on allocation • 1-6 VOLATILE attribute effect on allocation • 1-6 on pointer variables • 2-14

# $\overline{\mathbf{W}}$

WHILE statement
use for efficiency • 2-11
WITH statement
use for efficiency • 2-11
WRITE procedure
for sequential access • 4-12
on indexed file • 4-27

VAX PASCAL User's Guide Order No. AA-H485D-TE

# READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.					
Did yo	ou find errors in this manual? If so, specify	the error and	the page number.		
Please	e indicate the type of user/reader that you	most nearly re	present:		
	☐ Assembly language programmer				
	☐ Higher-level language programmer	<b>.</b>			
	<ul> <li>Occasional programmer (experienced</li> <li>User with little programming experienced</li> </ul>				
	☐ Student programmer				
	Other (please specify)				
Name	9	Date			
Organ	nization				
_	ot				
City _	Sta	ate	Zip Code		
•			or Country		



Do Not Tear - Fold Here



No Postage Necessary if Mailed in the United States

# **BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35 DIGITAL EQUIPMENT CORPORATION 110 SPIT BROOK ROAD NASHUA. NEW HAMPSHIRE 03062-2698

Cut Along Dotted Line